TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

SANTERI MÄKI-ÄIJÖ

ASSERTION-BASED FORMAL VERIFICATION OF A SOC MODULE SOFTWARE INTERFACE

Master's Thesis

Examiner:
Professor Timo D. Hämäläinen
Examiner and subject approved
30. November 2017

TIIVISTELMÄ

SANTERI MÄKI-ÄIJÖ: Väitelausekkeisiin pohjautuva järjestelmäpiirimoduulien ohjelmistorajapintojen muodollinen varmennus
Tampereen teknillinen yliopisto
Diplomityö, 48 sivua, 6 liitesivua
Elokuu 2018
Sähkötekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Sulautetut järjestelmät
Tarkastaja: professori Timo D. Hämäläinen

Avainsanat: System on a chip, väitelauseke, varmennus, muodollinen, RTL, SystemVerilog, SVA, ohjelmistorajapinta, kattavuus

Tässä työssä tavoitteena oli kehittää uusi väitelausekkeisiin pohjautuva muodollinen varmennusmenetelmä Nokia Mobile Networks SoC R&D:llä käytettävien järjestelmäpiirimoduulien ohjelmistorajapintojen varmennukseen. Nykyiset käytössä olleet simulaatioihin pohjautuvat menetelmät eivät olleet huomanneet kaikkia vikoja ohjelmistorajapinnoista, joten tarvittiin uusi menetelmä korvaamaan tai tukemaan jo käytössä olevia simulointimenetelmiä. Yrityksessä oltiin tutkittu väitelausekkeisiin pohjautuvan muodollisen varmennusmenetelmän soveltuvuutta erilaisten piirimoduulien varmennukseen, minkä pohjalta voitiin olettaa sen sopivan myös ohjelmistorajapintojen varmentamiseen. Tässä työssä selvitettiin menetelmän todellinen soveltuvuus ohjelmistorajapintojen varmentamiseen käytännössä.

Menetelmän käyttöä varten tuli luoda SystemVerilog-väitelausekkeita, joilla pystyttäisiin varmentamaan koko ohjelmistorajapinnan toiminnallisuus. Väitelausekkeiden luonti tehtiin kahdessa vaiheessa. Ensimmäisessä vaiheessa tavoitteena oli saada luotua väitelausekkeet, jotka pystyivät varmentamaan ohjelmistorajapinnan toiminnallisuuden. Tässä onnistuttiin hyödyntämällä sekä itse luotuja, että muodollisen varmennustyökalun tuottamia väitelausekkeita. Tässä vaiheessa ei kuitenkaan saavutettu tarpeeksi laajaa kattavuutta ohjelmistorajapinnan koko toiminnallisuudesta ja koodista, joten väitelausekkeita tuli parantaa.

Vaiheessa kaksi kohennettiin vaiheen yksi väitelausekkeita tuottamaan parempi kattavuus. Lisäksi pyrittiin saamaan aikaiseksi mahdollisimman yhtenäisen rakenteen omaavia väitelausekkeita helpottamaan assertioiden luomiseen käytettävän komentosarjan kehittämistä tulevaisuudessa. Tätä tavoiteltiin yhdistämällä muodollisen varmennustyökalun tuottamia sekä itse luotuja väitelausekkeita. Tässä vaiheessa onnistuttiin parantamaan väitelausekkeita kattamaan kaikki oleellinen toiminnallisuus ohjelmistorajapinnoista, mutta väitelausekkeiden lopullisesta rakenteesta ei saatu niin yhtenäistä, kuin toivottiin.

Työssä saatiin luotua uusi menetelmä ohjelmistorajapintojen varmennukseen. Kehitetty menetelmä ei yksinään pysty vielä korvaamaan käytössä olevia simulointeihin pohjautuvia menetelmiä, koska väitelausekkeiden luomista ei ole automatisoitu. Menetelmää suositeltiin kuitenkin käytettäväksi tukevana tapana ohjelmistorajapintojen varmennuksessa.

ABSTRACT

SANTERI MÄKI-ÄIJÖ: Assertion-based Formal Verification of a SoC Module Software Interface
Tampere University of Technology
Master of Science Thesis, 48 pages, 6 Appendix pages
August 2018
Master's Degree Programme in Electrical Engineering
Major: Embedded Systems
Examiner: Professor Timo D. Hämäläinen

Keywords: System on a chip, assertion, verification, formal, RTL, SystemVerilog, SVA, software interface, coverage

The goal of this thesis was to develop a new assertion-based formal verification method to verify SoC module SW interfaces used at Nokia Mobile Networks SoC R&D. Currently used simulation-based methods had been found to be ineffective in finding all bugs in SW interfaces. Therefore, a new method was needed to replace or support the current one. Previous studies had been made about the assertion-based formal verification method on different kinds of designs. The results implied that there is also potential to apply the method for SW interface verification. In this thesis, the suitability of the method for this task was examined in practice.

To use the method, SystemVerilog assertions verifying the whole functionality of the SW interface had to be created first. The creation of assertions was carried out in two phases. In the first phase, the goal was to create assertions that could verify the SW interface functionality. The goal was achieved by utilizing the assertions created manually and by a formal verification tool. However, the coverage of SW interface functionality and code achieved with the assertions was not good enough. Thus, the assertions had to be improved.

In the second phase, phase 1 assertions were enhanced to produce better coverage. In addition, the assertions should also be created with as uniform structure as possible to ease assertion generator script development in the future. To achieve these targets, the manually created assertions were integrated into the assertions created by the formal verification tool. By doing this, the whole relevant functionality of SW interfaces was covered. However, the structure of the assertions was not as uniform as was desired.

In this thesis, a new verification method for SoC module SW interface was produced. The developed method at its current state cannot replace the simulation-based methods, however, because the assertions are not generated automatically. Nevertheless, the method was suggested to be considered as an additional way to verify SW interfaces.

FOREWORDS

I want to thank Prof. Timo Hämäläinen, M.Sc. Mauno Tallgren, M.Sc. Juha Nousiainen and M.Sc. Antti Rautakoura for their great support during this thesis work. The start was difficult but thanks to good advice and guidance the work is now finished and I am even somewhat proud of it.

I also want to thank my fiancée for revising my English skills and supporting me during the thesis work.

Tampere, 06.06.2018

Santeri Mäki-Äijö

TABLE OF CONTENTS

APPENDIX A: Quantify assertion coverage report

ABBREVIATIONS

ABV	Assertion-based verification
AXI	Advanced extensible interface
CPU	Central processing unit
DUV	Design under verification
EDA	Electronic design automation
FV	Formal verification
HDL	Hardware description language
HW	Hardware
RTL	Register-transfer level
RW	Read-write
SoC	System on a chip
SV	SystemVerilog
SVA	SystemVerilog assertions
SW	Software
VHDL	Very high speed integrated circuit hardware description language
XML	Extensive markup language

1. INTRODUCTION

System on a chip (SoC) is an electronic system built on a single semiconductor chip. A SoC includes all required circuitry and components of a system. In addition to the required hardware (HW), a SoC often includes a processor capable of running software (SW) applications. This SW controls the SoC components and modules to perform desired actions. To enable these SW accesses to a module, it must contain memory elements, like registers. These memory elements can be accessed by a SW program running on a processor to monitor and configure how the module functions. A collection of these memory elements can be built into its own entity as a part of the SoC module, the SW interface of the module.

Verification is a process done before manufacturing a design to ensure that the design will function as specified. Verification is an important part of SoC development because fixing design errors and bugs is much more expensive and difficult after the chip has been manufactured. Therefore, verification takes a major part of the total time used in SoC development, even up to 80% [1]. New and improved verification methods are constantly developed to make the verification process more effective and faster.

This thesis was produced in the SoC organization of Nokia in Tampere. The specific goal was to develop a new way to verify SoC module SW interfaces used in the company. Previously, there had been bugs that had gone unnoticed by the simulation-based verification methods used for SW interface verification, which caused errors later in the development process. Therefore, the current verification methods were not enough. A different kind of approach was required either to support or replace the current methods to ensure the correct description of the SW interfaces.

Studies on assertion-based formal verification method usage on several types of designs had been made in the company with promising results [2]. It was also seen that formal verification (FV) had potential to work well for a design like the SW interface. Based on this knowledge, it was decided that an assertion-based FV method should be developed. A FV tool, which was already in use at the company, was utilized in the verification process. However, the assertions to be used as the model of correct design functionality had not been created for SW interfaces. The main goal of this thesis was to create the assertions and use them for the SW interface functional verification. The functionality and the design code of the SW interfaces should have been covered completely with the assertions. The assertions should also have had a structure that could allow them later to be generated with a script to automate the verification process. The script development is out of the scope of this thesis.

As the capability of assertion-based FV for SW interface was not known throughout, it was not sure that the developed method could actually achieve the set goals. Various reports provided by the FV tool would be used to analyze the functionality and code coverage of the design with the assertions. Both coverages should be 100%. In addition, information about the time consumed in the verification process and how simple the developed assertion-based FV platform is to use would be gathered. The results would be analyzed to see, if the developed platform could achieve these goals and how it compares with the currently used simulation-based methods. Based on the results, suggestions could be made, if and on what scale the platform could be utilized in SW interface verification.

The structure of this thesis is the following. Chapter 2 explains the assertion-based FV method and its key concepts. Chapter 3 presents the tools and methods used in this thesis. In chapter 4, the SW interface structure and its creation flow are described. The assertions were created in two phases. The first phase was about creating and using assertions for SW interface verification. Assertions, the used FV flow and the results of the first phase are presented in Chapter 5. In the second phase, phase 1 assertions were improved to have more uniform structure and provide better coverage. The enhanced assertions are then described in Chapter 6. Chapter 7 collects all the verification results of both phases including their analysis. Finally, the conclusions of the study are presented in Chapter 8.

2. ASSERTION-BASED FORMAL METHODS OF RTL VERIFICATION

Verification in digital hardware design process is about checking that a developed design matches its specifications or its model from a different abstraction level. It is done in parallel with the actual hardware design process before manufacturing the design. In practice, this means modeling the design and verifying it with various verification tools and methods.

There are multiple types of verification that are used when verifying a design. Timing verification, for example, ensures that the design meets its timing requirements. Power analysis is done to ensure that power requirements are met. To determine that the design will function as it is specified to function, functional verification is performed. These and other verification types require that the design has proper specifications. [3]

The modeling of the design can be performed on different abstraction levels. A higher abstraction level model is a less detailed description of a design compared to a lower level one. For example, a gate level model would include every logic gate, their connections and delays of the design whereas a higher register-transfer level (RTL) model considers the logic elements built from these logic gates and their connections in relation to the design clock cycle [4]. It is simpler to work on a higher abstraction level as there are fewer details to be considered. However, precision is lost when moving to a higher abstraction level, which could let some design flaws go unnoticed, if verification was done only on higher level. Timing issues, for instance, could be hidden in RTL model as there are no gate delays included. The functionality of a design is often described on RTL, therefore functional verification is feasible to be done on that level. It is the verification team's task to ensure that enough verification is done on low enough abstraction levels to find out all possible flaws.

A traditional and the most popular way to perform digital design verification is simulation. In simulation, different input stimuli combinations are driven to a design model. The model is monitored to see what it does with different inputs. The inputs can be pseudo-randomly generated, but even then, it is up to the designer to come up with all interesting input combinations. If all possible input combinations were tested exhaustively, time consumed during verification rapidly increases with the design size and complexity. [1, 5]

Formal verification is based on mathematically proving the correctness of a design. It computes what will happen in the design with all allowed input values instead of requiring them to be individually driven into the design and analyzing what happens as in simulation. FV covers all design states and functionality when properly implemented. In assertion-based FV, assertions are used to describe the design intent. The assertions are

checked with a FV tool whether they hold or not for the design under verification (DUV). FV performs best for relatively small designs as the larger the design is the larger its number of possible states, which quickly results in a major increase in time consumed during checking. [1, 5]

In this thesis, functional verification was performed with an assertion-based formal verification method on the RTL descriptions of SoC module SW interfaces. This chapter explains the key concepts of this kind of verification process.

2.1 Hardware description and verification languages

Hardware description languages (HDL) provide a way to describe logic circuitry in a text form. HDLs use a higher abstraction level than the transistor level to describe design functionality making it easier to design new systems. A common abstraction level used in SoC development is RTL. [4]

Two dominant HDLs, Very High Speed Integrated Circuit Hardware Description Language (VHDL) and Verilog, can be used to describe digital systems on different abstraction levels. VHDL and Verilog include similar features and structures and are nearly equally popular in the digital design industry. They both also provide an ability to create test benches to verify digital systems through simulations. VHDL was used for the development of SW interfaces discussed in this thesis.[4]

SystemVerilog (SV) is a unified hardware design, specification and verification language [6]. It is basically Verilog but with multiple extensions added to it. These extensions have created sublanguages under SV for different purposes in digital design. There is an Object-Oriented sublanguage used for verification, e.g., by Universal Verification Methodology (UVM) and a separate language for analyzing functional coverage. SystemVerilog Assertions (SVA) language is used to define assertions for verification. SVA was the language used for assertions created in this thesis. [7]

2.2 Constraints in formal verification

Formal verification analyzes that the design is functioning as it should with all possible input values. However, if the allowed input values of the DUV are not defined, the FV tool will analyze all values and combinations even though some of them can never occur in the design. To avoid this, constraints on the inputs can be defined. These constraints prevent the FV tool from computing false, unnecessary or unwanted input values and their combinations. This reduces verification runtime and ensures that valid inputs are used in design verification. It also allows creating specific situations or scenarios in the design to be verified. One method to implement constraints is defining SystemVerilog assumptions, which is explained in section 2.3.2. [7-9]

Constraints can cause problems in verification, if not properly defined. Implementing too few constraints on the inputs allows the FV tool to use invalid values while computing. This situation is called *under-constraining*. Conversely, there can also be too many constraints on the inputs. *Over-constraining* prevents some valid input values from being analyzed. This in its turn causes certain design behavior never being analyzed and can result in bugs not being found in the design. Constraints must therefore be carefully implemented so that the design is completely and correctly analyzed in the FV tool.[7, 8, 10]

2.3 SystemVerilog assertions

An assertion is a positive statement of what the design should do based on design specification. An assertion written with SVA is a precise and clear description of the design intent. If the assertion is false, it indicates an error in the design. SVA provides a faster and more understandable way to describe design intent than traditional HDLs like Verilog. Due to their unambiguous nature, SVA assertions fit well into FV. While the FV tool analyzes the design with possible inputs, the assertions are constantly checked whether they are true or not and, thus, verify the design functionality. If assertions cover all design functionality and the FV tool can analyze the design with all possible inputs, a design can be completely verified.[7, 9]

All assertions created in this thesis were SVA assertions. The following sections explain the structure and important features of SVA assertions.

2.3.1 Immediate and concurrent assertions

There are two main types of SVA assertions. Immediate assertions are non-temporal Boolean statements placed in procedural code. Immediate assertions resemble the condition of an "if" statement. They are executed like other procedural statements and are evaluated immediately when encountered in the code. They cannot consume simulation time. Immediate assertions can be edge or level sensitive to a clock or sampling signal. They do not include a property and are defined by the assert keyword only. Deferred immediate assertions are a special type of immediate assertions. These assertions evaluate their Boolean expression after every variable in it has settled down and not immediately when the assertion is triggered. This reduces the chances of glitches happening in the assertion evaluation. [7, 9]

The second main type of SVA assertions is the concurrent assertion. A concurrent assertion differs from an immediate assertion by being temporal. This means that the asserted expression can contain time consuming elements. This allows building more complex expressions and sequences to be asserted. A concurrent assertion always has a property that contains the expressions and sequences. The name concurrent comes from the fact

that concurrent assertions are executed in parallel with the design logic. They can be defined in *always* and *initial* procedures but also outside of any procedures. A concurrent assertion must have a clock edge defined to which it is sensitive. This is also called as the sampling edge on which the assertion evaluation starts. SVA concurrent assertions are very useful in FV as the FV tool can constantly check if the specified assertions hold for the DUV. In other words, concurrent assertions can describe design behavior in analyzable form for the FV tools. Every assertion created in this thesis was a concurrent assertion. [7, 9]

2.3.2 Concurrent assertion structure

A concurrent assertion is based on a *property* that describes some part of the design functionality. This functionality is expressed with Boolean expressions, e.g., that a certain signal in the design always has a value '1'. Boolean expressions can be combined to form *sequences*. In a sequence, the Boolean expressions are listed in linear order with respect to increasing time. A sequence is true, if all the expressions it consists are true on their defined clock ticks. Sequences can describe more advanced functionality than single Boolean expressions, like a register write or a bus handshake operation. A property may contain multiple sequences.[7, 9]

A property can be defined together with the assertion or on its own. If the property is defined separately, it allows the same property to be reused. A property can be included in either an assertion, cover or assume statement. If the property is used in *assert* statement, an assertion is created. If the property is part of a *cover* statement, it allows tracking of functional coverage of the design. If the cover statement is never true, it means that the functionality it describes never occurs in the design. *Assume* statements are used to constraint design behavior. They can, e.g., force certain signal to be high for two clock cycles whenever another signal has been high. In FV, assume statements are used to limit the input values and their combinations the FV tool uses when verifying a design. [7, 9]

Figure 1 presents the basic structure of a concurrent SVA language assertion. The meaning of concurrent assertion and the implication and $sampled seen in the figure, are explained in following sections. An assertion can have a label, which is basically its name, but it is not necessary. If no label is given, the tool used for assertion checking will give it a tool specific name. Here, the property is defined with the assertion statement, thus, creating an assertion. The assertion is evaluated on every positive edge of the clock, whenever the reset signal is low. When the assertion is evaluated, the Boolean expression(s) and sequence(s) are checked whether they are true or not. These expressions and sequences form the antecedent and the consequent of the assertion. The antecedent is checked first whether it is true or not. If it is true, the consequent is checked. If it is not true, the assertion is not evaluated. The assertion is true, or in other words holds, if the consequent is true after the antecedent triggered. Otherwise, the assertion is said to "fire" when it is false and an optional error message is displayed.

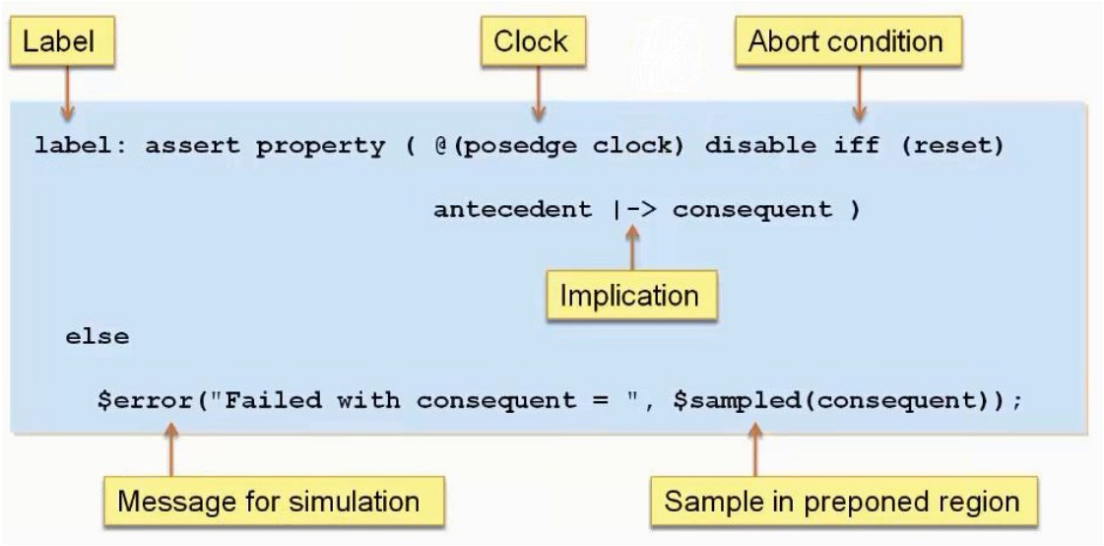


Figure 1: Concurrent assertion structure. [11]

Assertion can include multiple sequences that for multiple antecedent – consequent pairs. Assertions can also be sensitive to multiple clocks and their different edges and levels. Therefore, it is possible to create very complex assertions that can verify complex design functionality. However, it is advised that the assertions should be kept relatively simple and small. This improves the understandability of the assertions and their debugging. Complex design functionality should be verified by several smaller assertions. [7, 9]

2.3.3 Implication

Implication is a way to create sequential properties from sequences. The idea is that there is a precondition, an antecedent, and something that is evaluated after the precondition triggers, a consequent. This kind of structure was already seen in Figure 1. Between the antecedent and the consequent, is the implication construct. [7, 9]

There are two types of implication used in assertions. *Overlapping implication* is denoted with |->. In overlapping implication, the consequent checking is done right after the antecedent is true, which basically means on the same clock tick. In *nonoverlapping implication*, the consequent is evaluated on the next clock tick after the antecedent was true. Nonoverlapping assertion is constructed with |=>. [7, 9]

It is possible to include multiple implications in a single assertion property. The structure is always the same in that the left-hand side is the antecedent and the right-hand side is the consequent. Consequent of an implication can be the antecedent of another implication. The time when the consequent is checked can be modified by adding other time defining statements to the assertion. The consequent could for example be checked on the third clock tick after an overlapping implication's antecedent is true by adding ##3 after the implication construct. Adding *##[1:2]* after the implication construct would mean that the consequent is evaluated on the following two clock cycles. These time statements are

not directly part of the implication but allow more complex behavior to be described in the assertion with the implication. [7, 9]

2.3.4 SystemVerilog system functions

SV offers system functions and tasks that are usable in assertions and assertion control. These include *bit vector functions* that provide functions like *$onehot* to check that a certain bit vector only has a single bit high and *$countones* to count the number of bits that are high in the vector. Assertions can be controlled with certain tasks. *$asserton* and *$assertoff* enable or disable assertion checking. Separate assertions can be controlled with *$assertcontrol* and its different parameters. These control statements are not suitable for FV but are useful in simulations. [7]

The most used system functions in this thesis were the *sampled value functions*. These functions evaluate an integer expression on defined sampling edges (clock edges). If the integer expression is sampled to have values defined by the function, the function returns true. *$rose* function for example checks when the least significant bit (LSB) of sampled value rises from 0 to 1 between two sampling edges. When this occurs, the function returns true. The opposite situation of the LSB is evaluated with *$fell* function. To notice changes in the expression value being sampled*, $changed* function can be used. It considers all bits of the expression from current and past sampling edges. *$stable* function returns true if the expression value did not change during two consecutive sampling edges. To get a certain sampled expression value either *$sampled* or *$past* function can be used. $sampled returns the expression value sampled on the current sampling edge. $past in its turn returns the expression value sampled on some previous sampling edge, which is given as a parameter of the function. These functions provide handy ways to build assertion properties and improve their readability.[7, 9]

2.3.5 Witness and counterexample

While verifying a design, the FV tool checks whether defined assertions hold or not. If the assertion is true at least once, a *witness* could be generated. A witness is a report by the FV tool about an input sequence that resulted in the assertion being true. The report is different depending on the FV tool but at least the numeric values of the input sequences are included. A waveform of signals related to the assertion could also be presented. If the assertion fails at least once, a *counterexample* is created. A counterexample is the opposite of a witness as it reports an input sequence that caused the assertion to fail. Similar information as in witness is included in the counterexample. From these reports it is possible to analyze if the assertion is checking the intended behavior and, if the design does what it should. [7]

It is possible that the assertion is never evaluated during a FV tool check. This could happen due to the input values being too strictly constrained or some assume statements

preventing the behavior described in the assertion from occurring. There could also be a design bug or the assertion could be poorly written. The FV tool could also not have checked deep enough into the design state space to see the asserted behavior. Whatever the reason, it is much harder to find out compared to a situation where a counterexample was available.[7]

2.4 Formal verification tools

FV tools are EDA-tools that can verify designs with formal verification methods. The methods perform either equivalence or property checking. In equivalence checking, two versions of a design are compared, e.g. RTL and gate-level descriptions. The other version is already verified to be working correctly and another version is then compared to it. In property checking, correct design behavior is described with properties. These are then used to check if the actual design implements the same behavior. A good way to describe and check the properties is through assertions. The assertions can be created manually by a designer or they could be generated by the FV tool if the tool has an application for it. It is possible that a same FV tool includes both equivalence and property checking. [5]

FV tools typically offer additional applications on top of the property and equivalence checking. They can have applications dedicated to different error type, like X-state, checking or certain design structure, like register, checking. X-state checking analyzes, e.g., if the design has undefined signal values often caused by multiple drivers driving different values to the same signal. Register checks can verify, e.g., if the registers are accessible and that they have correct reset values. Different coverage calculation features are also common. FV tools might include assertion libraries or assertion generators for some standardized design parts like standard bus protocols.[12-14]

An example way to use the FV tool is to first read the design into the FV tool. This is done by providing the design for the FV tool in a format it understands, e.g., in RTL or IP-XACT [15]. The tool then analyzes it and creates an internal model of the design. This model is created through methods such as compilation and elaboration. Also, either a reference model or assertions describing the design intent are needed for the actual verification. After all necessary information is loaded to the tool, it runs the selected type of checks and possible other features. After the run, the tool provides reports with verification results, which can then be further analyzed by the designer.

3. TOOLS AND METHODS

This chapter describes the tools and methods used in this thesis. Reasoning, why the assertion-based FV was chosen, is explained. Also, the FV tool used for the verification is presented. However, the tools and flow used to create SW interfaces are described in section 4.2 due to them strongly being part of the SW interface description and partly a target of the verification.

3.1 Why assertion-based formal verification?

The goal of this thesis was to verify the functionality of a SoC module SW interface. Therefore, functional verification method was needed. Functional verification determines if the design matches its specifications or not [3]. FV is static functional verification, which means it is a feasible method for the task [16]. The size and complexity of a SW interface were considered not to be too great for achieving tolerable FV check runtimes. Due to this, FV was a valid option for SW interface verification.

Proper and comprehensive input stimuli for the SW interface functionality would have been difficult to create. Also, a new set of input stimuli would have been required for every new SW interface. Simulations were therefore not a good enough solution for complete SW interface verification. With FV, constraints are utilized to limit analyzed input sequences. The FV tool analyzes the design behavior with all input sequences not ruled out with the constraints. As long as the constraints are properly defined, this eliminates the need to create input stimuli, which further backed up choosing FV as the verification method.

FV requires either a reference model or design to be compared with the DUV to see, if it is working as specified. The SW interface specifications clearly described the possible functionality a SW interface could have. The functionality also seemed to be feasible to be described as assertions. It was not certain that it would have been possible to create assertions for all functionality. However, previous knowledge and experiences ([2]) implied that assertions would work well in this situation. In addition, assertions for part of the SW interface functionality, like for the bus between CPU and the SW interface, could have been generated with a capable FV tool. Due to these reasons, assertions were chosen to be the reference of the correct design functionality for FV.

3.2 Onespin

The FV tool used for verification in this thesis was Onespin 360 DV Verify [14]. It was already used in the company and had an appropriate set of features for SW interface verification. A brief introduction of the used features is given in this section.

The tool could perform assertion-based FV on digital designs. For this, the design files and optional files that contained the assertions are required. In this thesis, the design files were the design source code in VHDL and IP-XACT format description of the design in XML-file. The tool performed elaboration and compilation to produce a model of the design that it could then run FV checks on. During elaboration, the tool determines the design top level and forms the hierarchy of all other design elements inside it [17]. In compilation, the design elements are compiled [17]. Assertions were provided as a separate SVA file. The tool could generate assertions for checking certain bus protocols, including the one used in the verified SW interfaces. This feature was utilized for the bus verification. The assertion file had to be bound to the design module instance created by the FV tool [16]. This way, the assertions could be checked by the tool on the correct design.

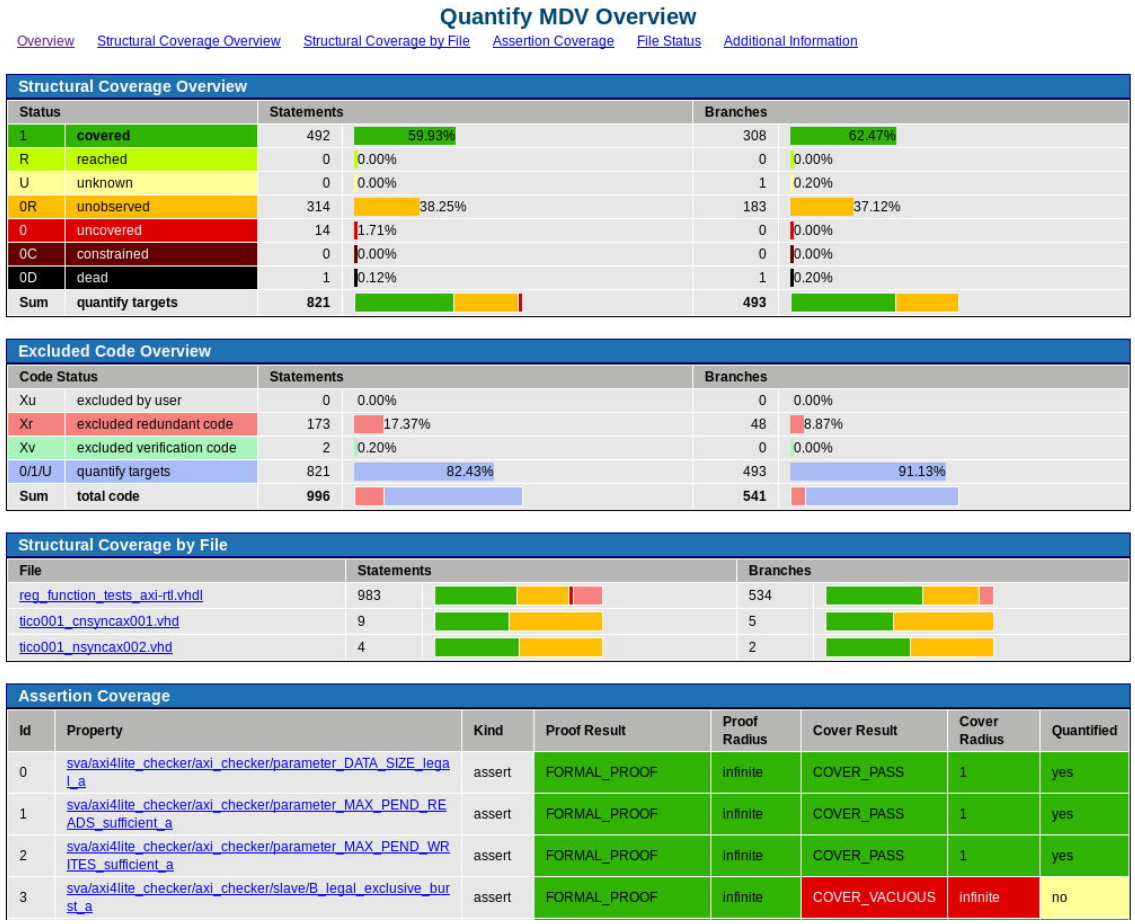
After the actual assertion checking, the tool provided various reports to analyze the results. These reports were one of the key ways to confirm that a part of SW interface functionality, and/or the assertion created to verify it, worked as was intended. The tool created witnesses and counterexamples when possible. Both could be opened to see a waveform of related signals and their values few clock cycles before and after the assertion condition was fired. In addition, the assertion and design source code could be viewed. The tool noted in the source codes what values the parameters had and which line was active at a certain clock cycle. If no witness or counterexample could be created, the tool provided information about why they could not be created. A detailed explanation how these features were used in SW interface verification is given section 5.2. Figure 2 shows a snapshot of a witness report waveform view for one assertion, which checks that register *reg_basicreg_s* value changes are immediately visible in its output port *basic_reg_out*.

Figure 2: Witness waveform view.

Another feature used to analyze completeness of assertions and the design verification was a formal code coverage feature called Quantify. It analyzes the design code statements and branches to see if they are observed by assertions currently used under current constraints. *Observed* means that the statement or branch is covered by at least one assertion. The tool also analyzed if the code statement or branch was reached in the assertion check ran by the tool. For the statement or branch to be fully covered, it had to be both

reached and observed. This feature was used after some assertions were created and already checked in the tool to confirm that the assertion checked everything it was supposed to. [10]

An example of Quantify report overview is presented in Figure 3. The report shows the design and assertion files used in the assertion check, percentages of how well the design code is covered and information about each assertion included in the check.



Quantify MDV Overview

Overview | Structural Coverage Overview | Structural Coverage by File | Assertion Coverage | File Status | Additional Information

Structural Coverage Overview

Status		Statements		Branches	
1	covered	492	59.93%	308	62.47%
R	reached	0	0.00%	0	0.00%
U	unknown	0	0.00%	1	0.20%
0R	unobserved	314	38.25%	183	37.12%
0	uncovered	14	1.71%	0	0.00%
0C	constrained	0	0.00%	0	0.00%
0D	dead	1	0.12%	1	0.20%
Sum	quantify targets	821		493	

Excluded Code Overview

Code Status		Statements		Branches	
Xu	excluded by user	0	0.00%	0	0.00%
Xr	excluded redundant code	173	17.37%	48	8.87%
Xv	excluded verification code	2	0.20%	0	0.00%
0/1/U	quantify targets	821	82.43%	493	91.13%
Sum	total code	996		541	

Structural Coverage by File

File	Statements	Branches
reg_function_tests_axi-rtl.vhdl	983	534
tico001_cnsyncax001.vhd	9	5
tico001_nsyncax002.vhd	4	2

Assertion Coverage

Id	Property	Kind	Proof Result	Proof Radius	Cover Result	Cover Radius	Quantified
0	sva/axi4lite_checker/axi_checker/parameter_DATA_SIZE_legal_a	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
1	sva/axi4lite_checker/axi_checker/parameter_MAX_PEND_READS_sufficient_a	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
2	sva/axi4lite_checker/axi_checker/parameter_MAX_PEND_WRITES_sufficient_a	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
3	sva/axi4lite_checker/axi_checker/slave/B_legal_exclusive_burst_a	assert	FORMAL_PROOF	infinite	COVER_VACUOUS	infinite	no

Figure 3: Quantify report overview.

4. SOFTWARE INTERFACE OF A SOC MODULE

This chapter describes the purpose of the SW interface in a SoC module and how it is created. In addition, a detailed description of the interface's register functionality, the importance of the interface for SoC modules and how its functionality is currently verified are presented.

The SW interface of a SoC module is a collection of physical registers contained in a separate block on the chip. The SW interface can be accessed by software running on a central processing unit (CPU) through a bus with read and write operations. Figure 4 is a simplified illustration of SW interface structure with all the key elements shown. [18]

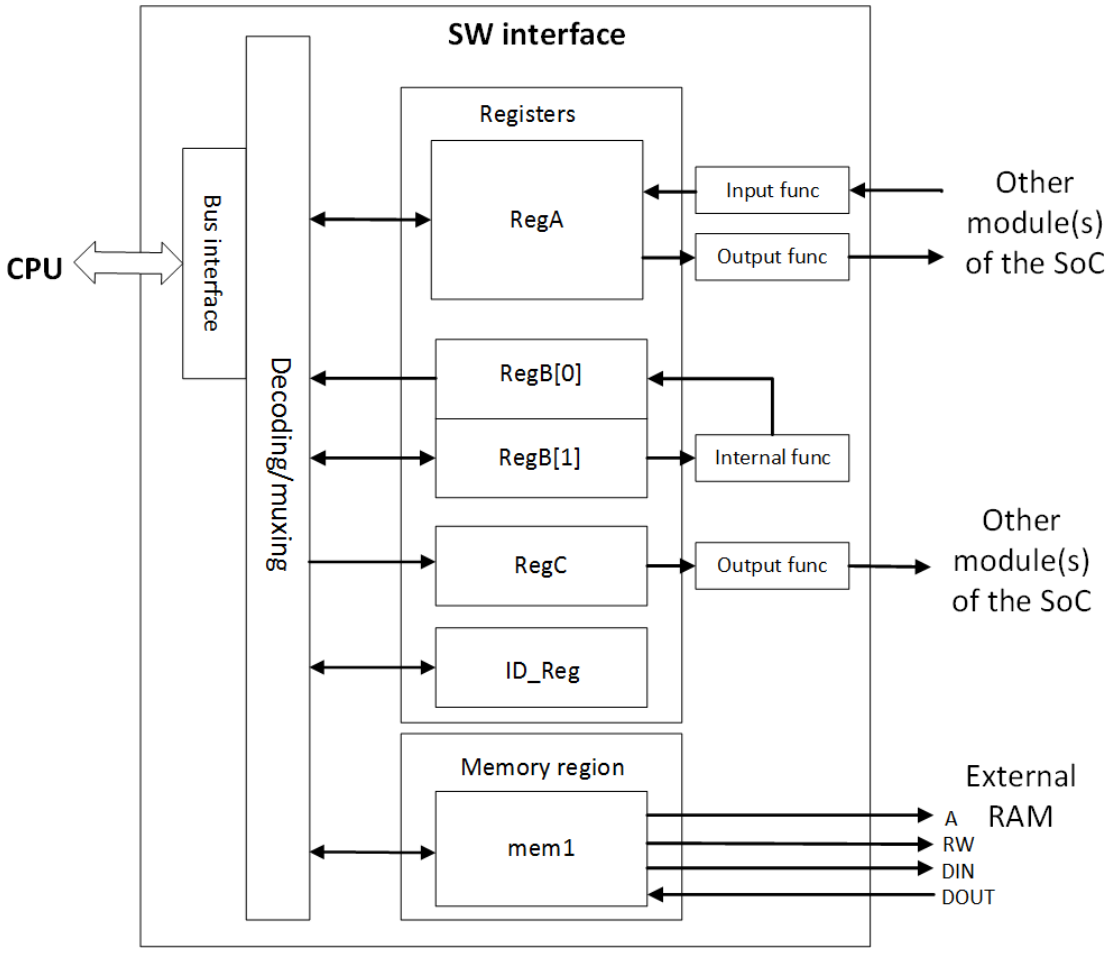


Figure 4: Illustration of a SW interface.

On the left in the figure is the bus interface through which the CPU connects to the SW interface. The registers in the SW interface can be read and written through this bus interface. Decoding/muxing element takes care of data reading from or written to the correct register. The registers can have different types. RegC, for instance, in the figure is a write-only register since its value can only be written through the bus interface. Register RegB

field ‘0’ is a read-only type because it can only be read by the bus. Different register types are explained in section 4.3.1.[18]

The SW interface can contain a special functionality to access and use the registers independently from the bus interface. These register functionalities are defined as functions, which are shown on the right side of the Figure 4. The functions can, for example, create additional I/O ports that allow other HW modules to be connected to the SW interface or make a register value depend on the value of other register. The functions will be explained in detail in section 4.3.2. [18]

In addition, it is possible to define memory regions in the interface mapped to the register address space. Each memory region has its own RAM input and output ports that are connected to a RAM block located elsewhere in the SoC. [18]

4.1 SW interface functions

The SW interface has a couple of key functions in a SoC module. The interface can be used to control certain parts of a SoC module as its functionality is possible to be tied to the register contents. Part of the module, for example, could be enabled or disabled depending on a certain register value in the interface. Also, parameters for the module can be given through the register. [18]

Another function for the SW interface is that it can provide status information of the SoC module for SW. HW can be built to modify certain register content e.g. to describe different errors having occurred in the module. This kind of status information from the module is vital for SW, e.g., to act to solve an occurred error. [18]

The third important application for the SW interface is the handling of interrupts generated by the SoC module. SW interface registers can be connected to HW so that they allow the SW to notice, handle and clear interrupts generated by the HW. SW can read interrupt status register(s) to get information, which interrupt is active, and then do required operations to handle the interrupt. After the interrupt is handled, SW can clear the active status of the interrupt by writing to a certain register. [18]

The described SW interface functions are the most common use cases. However, it is possible to use a SW interface for a wide range of specialized tasks that do not belong to the presented use cases. A SW interface can include many registers that have numerous different use cases. This can make an interface a complex and large block. However, the structure of SW interfaces is always the same. For this reason, it has been possible to automate the creation process to produce SW interfaces without writing any RTL code. [18]

4.2 Automated creation process of a SW interface

Before any design is created, a detailed specification of the requirements for the product should be made. For a SW interface, the number of registers, their size and type of the registers must be defined. The bus protocol used to access the SW interface must be decided. The possible constraints for power consumption and resource usage must be met by the interface. The specification stage is important because changes to the specifications made after the SW interface has already been created also require changes to the interface. This could mean complete recreation of the interface.

There exists a special flow for creating the SW interfaces. The flow consists of three steps, which are explained next. [18]

4.2.1 Step 1: Defining a SW interface

After the specifications have been agreed on, the creation flow starts with defining the interface registers in an MS Excel spreadsheet. The spreadsheet contains separate sheets with different information. A cover sheet is used to document changes made to the spreadsheet. Every new version of the spreadsheet will have its own version number, date it was made, name of the person who made the changes and a brief description of what was done. Then, another sheet has all additional parameters to be included as VHDL generics in the interface, if those are needed. The most important sheet is the one where every register of an interface is defined. An example definition of a register is shown in Table 1. [18]

Table 1: SW interface register definition in MS Excel

group	address	register	field	start	stop	dim	rw	default	hdl_path	description
reg_function_tests										
default										
	0x04	RegA				1				
			Bits	31	0		RW	0x0		[Input Function: PORT Basic_reg_in] [Output Function: PORT Basic_reg_out]

Table 1 shows the definition of a 32-bit register named *RegA* that has a single register field called *Bits*. Its direct address is 0x04, which is used by the SW to access this register through the bus. The register is part of an interface called *reg_function_tests* and *default* register group. Only one copy of this register is created as its dimension is 1, which is

defined in the dim field. If the dimension is bigger than 1, multiple instances of the register are created with an indexing number added to their name and with consecutive addresses. The register type is defined in the rw field and its default value in the default field. For the register in the figure, the type is *read-write (RW)* and default value is 0. HDL_path for a register is not defined in the spreadsheet but later in the flow. The HDL_path is the RTL description path from top level all the way down to the register, which can be used in verification to directly access the register. The description field is for describing the register and for defining functions for the register. These functions define additional functionality for the register and are explained in detail in section 4.3.2. For the register in Table 1, an output function *PORT Basic_reg_out* is defined, which means that the register will have its own output port named *Basic_reg_out* in the interface. A register definition can also have a separate field for comments about the register. [18]

A register can include multiple fields with different type, width and function but there are some constraints for their values. The maximum register width is the width of the CPU bus as it must be possible to write the whole register with a single write operation. The number of fields a register can have is the width of the register since a single field must be at least 1-bit wide. Each field of a register can have only one type but separate fields inside a register can have different types. These types are explained in section 4.3.1. [18]

It is not required that all registers of a SW interface must be defined in the spreadsheet. Although if a register is not defined in the spreadsheet, it must be manually added to the files created by this SW interface generation flow.

4.2.2 Step 2: Creating IP-XACT representation of a SW interface

The step following the MS Excel spreadsheet definition is to generate IP-XACT standard form representation of the spreadsheet content in an XML-file. This is done because the IP-XACT format file is suitable input for many EDA-tools, including the tools used in this creation flow. [15, 19]

In addition, information of the bus protocol is added in this stage to the XML-file. There are several choices for the bus protocol but AXI4Lite is the most common selection [20]. An EDA-tool provided by Magillem is used for this step in the creation flow but any other tool with the same features could be used. Lastly, an in-house script for the tool is used to produce a text file description of the just created XML-file, which helps in the final step. [18]

4.2.3 Step 3: Generation of RTL description of a SW interface

The final step in the creation flow is to generate an RTL description of the SW interface in VHDL format. An in-house register generator script is used to generate the actual RTL

code from a text file created in the previous step. The script parses the design information from the text file and produces a VHDL file as a result. When the VHDL file is ready, the creation process is finished and the SW interface is ready for synthesis.[18]

4.3 Register functionality in a SW interface

As seen in sections 3.1 and 3.2, the registers in the SW interface block are not just for storing data that can be read and written. From the SW point of view this could be the case as it can only read or write different registers in the interface through the bus. However, additional functionality can be added to the interface. This functionality is declared together with the register type and function definitions in the register definition table.[18]

4.3.1 Register types

All registers in the interface all have a type. The type defines if the register is writeable, readable, clears its bits when written or read, and so on. The type is defined for every register field individually, so a register can have fields of different type. All possible register types are shown with a brief description in Table 2. [18]

Table 2: SW interface register types

Register type	Description
RW	read-write
RO	read only
RC	read to clear all
RS	read to set all
WO	write only
WC	write to clear all
WS	write to set all
WRC	normal write, read to clear all
WRS	normal write, read to set all
W1C	write 1 to clear all
W1S	write 1 to set all
W1	write once
WO1	write only (once)
WOC	write only, clear all
WOS	write only, set all
WSRC	write sets and read clears all
WCRS	write clears and read sets all
W1CRS	write 1 clears, read sets all
W1SRC	write 1 sets, read clears all
W0CRS	write 0 clears, read sets all
W0SRC	write 0 sets, read clears all
W1T	write 1 to toggle

Register type	Description
W0T	write 0 to toggle

Different register types are used for different purposes. A RW register is flexible and has the most use cases of the register types as it is compatible with almost all functions a register can have. Other register types are used when the register has a specialized purpose, e.g., a RO type could be defined when the register contains a constant parameter that should therefore never be rewritten or if the register value can only be modified by HW. [18]

4.3.2 Register functions

There are three main function types of which a register can have functions defined. All functions are defined by adding *[<function_type> Function: <function>]* to the description field of the register. [18]

Input functions describe functionality that control what is written into a register. *PORT* -function is used to create an input port for the register, which can be used to drive data to the register. An example definition is shown in Table 3. [18]

Table 3: Input function definition

address	register	field	start	stop	dim	rw	default	hdl_path	description
0x04	BasicReg				1				
		Bits	31	0		RW	0x0		[Input Function: PORT Basic_reg_in]

It is optional to provide a name for the input port. A name derived from the register name will be created if no name is defined in the function declaration. The input port function can be combined with other functions.[18]

EDGE and *LEVEL* input functions are used set or clear all register bits on certain edge or level of the input port. The *LEVEL* functions may also add an enable signal that needs to be either high or low for the input port value to be written into a register. This enable can be either an additional input port or a different register field. These functions are used, e.g., for interrupt status registers as they can capture interrupt events coming from the module through the input port. An example declaration of a register with *LEVEL* function is shown in Table 4. This register will update its value from the input port *REG1_in* when *ENABLE_REG* field *REG1_en* has the value '1'.[18]

Table 4: Level function definition

address	register	field	start	stop	dim	rw	default	hdl_path	description
0x00	REG1				1				
		Bits	15	0		RW	0x0		[Input Function: LEVEL1 ENABLE_REG.REG1_en; PORT REG1_in]

Other input functions include functions that compute a register value with an *OR*, *AND* or *XOR* operation of two registers. The rest of the input functions are used for adding delay or synchronizing stages to the input.[18]

Output functions define how the register (field) value is used. With *PORT* function, a dedicated output port can be created for the register. This function can be defined like the input *PORT* function. Only the function type must be defined as *Output*. A name for the port can be defined but it is not mandatory. A name derived from the register will be used if no name is specified. Like for the input functions, the *PORT* function is usable with other output functions. [18]

A *DELAY* function must be combined with *PORT* function. *DELAY* function adds a specified delay after which the register value is visible in the defined output port. *DELAY* function does not affect the CPU bus. An example definition of a register with output delay is presented in Table 5. The delay for this register's output is two clock cycles.[18]

Table 5: Output function example

address	register	field	start	stop	dim	rw	default	hdl_path	description
0x50	DelayReg				1				
		Bits	3	0		RW	0x3		[Output Function: PORT DelayReg_out; DELAY=2]

Other output functions include *OP* functions that add a 1-bit output signal or port to the register. This signal or port holds the result of a Boolean operation performed on the register bits. Clock synchronization functions are used to synchronize the register output to a clock signal that is not the clock used by the SW interface. Finally, there are *STROBE* and *PULSE* functions to produce 1-bit signals that trigger when the register value changes or a new value is written to the register.[18]

The third main function category is the **internal functions**. Internal functions define functionality for a register field inside the SW interface. The most common internal function is the *ID* function. *ID* function is used to create a register that holds identity information

of the SW interface block that contains the register. The *ID* register holds multiple description lines of information that can be read with consecutive read operations from the register. An example *ID* register definition is shown in Table 6.[18]

Table 6: Internal function example

address	register	field	start	stop	dim	rw	default	hdl_path	description
0x0	Id				1				Identification register
		ID	31	0		RW	32'h55		[Internal Function: ID] Value 0 : 32'h55555 Synchronization word Value 1 : 32'h000239 Component code Value 2 : IDVersion-Number_C Value 3 : 32'h0 Build number Value 4 : MySize_C *Module parameter #1*

Other internal functions include *BUS_ERROR* that counts illegal bus accesses to an interface and *WIRE* that creates a read-only register without any flip-flops. *CONSTANT* function defines that a register contains a constant value. *COMBO_** functions indicate a multi-address register, which can be accessed through several addresses but there is only one physical register implemented. The '*' in the *COMBO* function can be either *STATUS*, *SET*, *CLEAR* or *TOGGLE*.[18]

There are few function types that do not fit clearly into one of the three main function categories. These are the **miscellaneous functions**. *SET* and *CLEAR* functions are used to allow register content to be set or cleared when a defined condition is valid. The condition is that either a port, a register field or a certain bit in a register field has a certain value. [18]

ENABLE function defines an additional enable condition for register writing. The enable condition must be true for the CPU to be able to write a new value to the register. The condition can have ports and register fields compared to certain values or other ports and register fields. It is also possible to include AND, OR and NOT expressions to the conditions. However, it is not possible to include both port names and register fields in the same condition expression. Table 7 shows an example definition of *SET* and *ENABLE* functions that both have a similar condition where a port value needs to be a certain value, here '1'.[18]

Table 7: SET and ENABLE function example

address	register	field	start	stop	dim	rw	default	hdl_ path	description
0x100	REG				1				
		high	31	16		RW	0x0		[Set Function: PORT SetPort = '1']
		low	15	0		RW	0x123		[Enable Function: PORT EnablePort = '1']

The last miscellaneous function is the *memory* function, which is used to implement register address space mapped memories. These memories are accessible by CPU through the SW interface bus with the same write and read operations as normal registers. The memory itself is located outside of the SW interface and is connected to the interface through its input and output ports that are mapped to the SW interface registers.[18]

In total, there are about 30 different functions that can further be defined with different conditions and combined with each other. The number of functions a single register can have is not limited. The designers can define as many of them as they like for a register. However, not all functions work together and will cause errors. Which functions may be used together is up to the designer to know. The register type must also be suitable for the functions defined to work. For example, a register with *CONSTANT* function should not have a type that allows the register content to be changed, like *RW* that allows bus write operations.[18]

4.4 Verification and usage of the SW interface

The registers in SW interfaces are partly verified during the SoC verification process. Simulations that make read and write operations with random data are made to check that the register is accessible. However, the special functionality defined with register functions is left untouched as they are difficult and time-consuming to verify. The tools and scripts used to create the SW interface do verify that the defined interface has registers with correct widths, addresses and valid port, register and register field names in the function description [18]. These do not check that the generated VHDL code implements the described function as it should be implemented.

Step 2 of the creation flow presented in section 4.2.2 is verified by the EDA-tool provider. It is the in-house register generator script, which creates the RTL implementation that is not fully verified. Test runs have been made with the creation flow to see that it is working for the most common register types and functions. However, not all possible register definitions have been verified. This means that it cannot be said that a SW interface created with the flow is 100% correct.

SW interfaces created with the described flow are used widely in SoC projects in the company. There have been cases where a created interface has had errors and the causes had been hard to find. A FV tool that supported assertion-based verification was already in use in the company. If assertions were created for all functionality, implementation of a SW interface could be verified. Efforts made to create the assertions are described in Chapters 5 and 6.

5. PHASE 1: CREATING ASSERTIONS TO VERIFY SW INTERFACE FUNCTIONALITY

This chapter presents the first phase of the thesis work. At the end of this phase, assertions for all SW interface functionality added with the register functions should have been created. During assertion development, possible bugs found in the SW interface generator would be reported. Also, it would be known how well the assertions covered SW interface functionality and if the assertion-based FV platform being developed is suitable for the task as assumed.

5.1 Verification of register types

The formal verification tool automatically created basic read and write operation SystemVerilog assertions for the used AXI4Lite bus protocol [14]. These assertions were used to check if the register types are generated as mentioned in the register generator specification [18]. It was also analyzed how the different register types of the SW interface are covered by automatically created assertions. The register types were verified first to make sure that they function correctly before starting to create assertions for the register functions.

The design under verification (DUV) was a SW interface, which included registers with a single field, generated with the creation flow described in Chapter 4. Each register had a different register type used in their field. This kind of approach was chosen to clearly separate the checking of a single register type from the others. All possible register types were included in the DUV.

Some problems were encountered when running assertion checks on the DUV. At first, all register read and write assertions failed. Examining the FV tool's counterexample report revealed that the read and write address signals of the AXI4Lite bus were not completely constrained to function as they did in the interface. The address signals were, by default, 32-bits in the formal tool but the interface only used 12-bits. In addition, register addresses in the interface were defined with four bytes offset from each other starting from 0. This meant that the lowest two bits of the address signals were always 0. Neither of these characteristics of the interface were told to the formal tool through constraints, which allowed the tool to wiggle bits that it should not and cause the assertions to fail. The importance of restricting the input values to possible input values of the design was noticed through these problems. Otherwise, assertions might fail even though the design is working as it was intended. From this point on, special care was used to ensure that under-constraining would not cause errors in verification.

Table 8 presents the results of register type verification with assertions. It shows if the assertions created for the register held and if the created VHDL implementation of the register matched with register specifications. It is also shown if a witness or counterexample could be created for the assertion. The register types the interface can have are read-write (RW), read-only (RO), read-clear (RC), read-set (RS), their write operation counterparts and different combinations of them.

Table 8 : Register type verification results

Register type	AXI4Lite checks hold	VHDL matches type	Witness/counterex
RW	yes	yes	yes
RO	yes	yes	yes
RC	no	yes	yes
RS	no	yes	yes
WO	yes	yes	yes
WC	yes	yes	yes
WS	yes	yes	yes
WRC	no	yes	yes
WRS	no	yes	yes
W1C	yes	yes	yes
W1S	yes	yes	yes
W1	yes	yes	yes
WO1	yes	yes	yes
WOC	yes	yes	yes
WOS	yes	yes	yes
WSRC	no	yes	yes
WCRS	no	yes	yes
W1CRS	no	yes	yes
W1SRC	no	yes	yes
W0CRS	no	yes	yes
W0SRC	yes	no	yes
W1T	yes	yes	yes
W0T	yes	yes	yes

Few register types did not get correct VHDL implementation generated by the generation script. Write-clear and write-only clear types (WC, WOC, WO1) caused errors in the script and blocked the generation of the whole interface. W0SRC -type caused no errors in the script but it did not get correct VHDL implementation either. It was created as a basic RW-type register. Based on these findings, the script was fixed to correctly support and implement these register types.

The assertions that did not hold had either RC or RS functionality. They failed because the assertions considered registers being able to only change their value through bus write operations. RC and RS type registers modify the register content when read, which causes the written and read values not to match and the assertion to fail. However, the correct functioning of the register types was possible to be fully verified from counterexample report waveforms and by looking at the generated RTL implementation.

Figure 5 shows the waveform view of counterexample report created for a RC type register. The assertion fails due to the read register value (RDATA) not matching the previously written value (READCLEARTEST_B0). This does mean that the register content is cleared without a bus write operation, which implies that the register is working as it should. It was confirmed by viewing the RTL implementation that the register functions correctly.

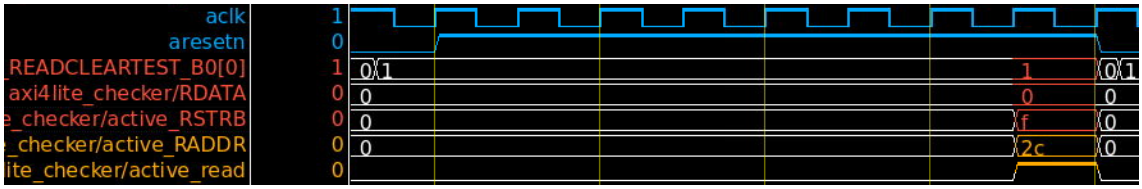


Figure 5: Read-clear register counterexample.

Even though automatically generated assertions did not hold for all register types, the types could be verified to be working as intended by examining the witness and counterexample reports and the VHDL implementation of the SW interface. Few bugs were found and fixed in the interface generator script and it was now functioning correctly for the register types. However, since the FV tool automatically created assertions for the register did not work for all types, they could not be used as such to achieve 100% functional coverage. Improvements or completely new assertions would be required to fully cover the bus read and write operations. This problem was decided to be solved later, when assertions for the register functions were created.

5.2 Verification of register functions

This section explains how an assertion for a SW interface function was created and how both the assertion and the function were verified to be correctly implemented.

5.2.1 Creating an assertion

The first step in assertion creation was to understand the specification of the function the assertion was going to verify. Without fully knowing what the functions should do, a proper assertion for it could not be created. If the specification was misunderstood, time would be wasted on debugging an incorrectly designed assertion. Also, it might be possible for a wrongly implemented function to get through verification, if it was not noticed

that the assertion did not verify it correctly. This step was executed carefully to prevent these cases from occurring.

The next step was to generate a SW interface with registers that have the functions to be verified. Similar structure as with the register types, one function per one register (field), was used in the interface. Two to three new registers with new functions to verify were added at the same time. Assertions for these were then created and used to verify the registers before adding new ones. This approach was used during the whole phase 1.

After the preliminary steps were done, the assertions could be created. The assertions were created with SVA language. A separate file for the assertions was used for the manually created assertions to clearly separate them from the assertions generated by the formal tool. An assertion was designed to check that a register, I/O port or an internal signal of the SW interface changed its value or displayed a correct value as defined by the function. Real values of the registers, ports and signals were considered in the assertions. The values and from what point of time they were considered varied greatly in the assertions.

Assertions were created individually. One assertion had to be complete before starting to create a new one. An assertion was complete when it could verify the functionality defined with a register function it should verify. This was ensured with verification tasks as described in section 5.2.2.

5.2.2 Verifying a function

To verify a function, a SW interface containing a register with the function and an assertion for the function were created. Assertion and design files were read into a FV tool. A configuration file for the tool was also included. The file included options and information about the DUV so that the tool knew how to handle it and what automatic checks and constraints it should create. These files were then compiled and elaborated with the desired options to load the design into the tool.

The tool provided formal assertion proving methods, which were used to check whether an assertion holds or not. The assertions could be checked separately, in selected groups or all at once. Checking fewer assertions at the same time reduced the runtime of assertion checks. This option was used often during the verification process.

Figure 6 shows a capture of assertion check report screen of the FV tool. The figure shows the name of checked assertion(s), if the assertion was proved or not and the check runtime. Other information, such as validity of assertion check, could also be seen in this screen. If the assertion could be proved to either hold or fail, a witness or counterexample was produced.

Name	Proof Status	Witness Status	Runtime
...tion_tests_STROBEREG_write_reg_okay_a	hold	pass (3)	00:00:01
...nction_tests_WIREREG_BIT_access_reg_a	hold	pass (4)	00:00:07
sva/register_check/my_BASICOUTPUT_a	hold	pass (2)	00:00:01
sva/register_check/my_CLEARPORT_a	hold	pass (1)	00:00:01
sva/register_check/my_CLEARPORTINDEX_a	hold	pass (1)	00:00:01
sva/register_check/my_CLEARREG_a	hold	pass (3)	00:00:01
sva/register_check/my_CLOCKSYNC_a	hold	pass (6)	00:00:01
sva/register_check/my_CONSTANT_a	hold	pass (1)	00:00:01
sva/register_check/my_DELAYREG2_a	hold	pass (4)	00:00:02

Figure 6: Part of assertion check report screen.

Even if the assertion did hold and a witness was created, further analyzing had to be done to see if the assertion was verifying correctly the function it should verify. From the report screen presented in Figure 6 it was possible to open the witness or counterexample report. In the opened window, the assertion source code and a waveform viewer were presented. Based on the waveforms and signal and port values, it was most of the time clear whether the assertion and the function were working as intended.

In addition to reports provided by the FV tool, the VHDL implementation of the SW interface was analyzed manually by looking at the code. This code viewing helped to understand whether an assertion was failing because of a faulty assertion or incorrect implementation of the function. The VHDL implementation was viewed for every function but it was used more as an additional verification method to the FV tool reports.

The coverage of SW interface VHDL implementation by the assertions was also checked as one step in function verification. The FV tool provided a feature, Quantify, that could analyze what parts of the interface code were considered or active when proving an assertion [10]. With this feature, it was possible to see if an assertion truly verified the RTL implementation of the function it should verify. This coverage check was run after an assertion was already holding and seemed to be verifying what it should to see if the register code was fully covered.

Verification tasks explained in this chapter were performed until it was sure that an assertion was correctly composed to verify the function implementation it was supposed to verify. Bugs found in the function implementations would be reported further to be patched. The goal was that every function of SW interface would have an assertion created to verify it. Section 5.4 describes how well this goal was actually met, including other results of this phase 1.

5.3 Example assertions

This section presents three register function assertion examples. The assertions selected include functionality that is common in SW interfaces. They also consist of most of the different features used in the created assertions.

The first assertion described is an assertion for *input PORT* function. This function creates an additional input port to the register, which can be used to write new data into the register [18]. The assertion property is shown in Figure 7, where the input port is *BASICREG_IN* and the register is *reg_BASICREG_s*.

```
// Basic register with named ports. Input port assertion.
property BASICREG_port_in_p;
     @(posedge ACLK) disable iff(!ARESETn)
       $changed(BASICREG_IN) |=>
       reg_function_tests_axi.reg_BASICREG_s ==  $past(BASICREG_IN);
endproperty
```

Figure 7: Assertion for input PORT function.

The assertion evaluates on positive clock edges when reset is not active. The $changed SV function is used to notice the change in port value, which is the antecedent. Nonoverlapping implication |=> means that the consequent checking is done on the next clock tick. In the consequent the register value is compared to the port value from the previous clock tick. If the function is correctly implemented to the register, the compared values should match and assertion will hold. This function was one of the easiest to create an assertion for and was very similar to *output PORT* function assertion.

The second assertion example is for *output DELAY* function. The *DELAY* function was always used with a PORT function to create an output port for a register, which would show a register value after a defined delay [18]. The delay is measured in clock cycles. The assertion is displayed in Figure 8, where the *DELAY_REG2_OUT* is the output port and *reg_DELAYREG2*_s is the register.

```
// Delay function for output port
property DELAYFUNC_p(int delay);
  @(posedge ACLK) disable iff(!ARESETn)
     $changed(reg_function_tests_axi.reg_DELAYREG2_s) |->
     ##delay reg_function_tests_axi.reg_DELAYREG2_s ==
     reg_function_tests_axi.DELAY_REG2_OUT;
endproperty
```

Figure 8: Assertion for output DELAY function.

For this assertion, a property with a parameter was used. The parameter was an integer number that defined how long a delay there should be until the register value is visible in the output port. This allowed the same property to be used for all delay functions with

different delays. The delay is implemented with syntax *##delay*, where the delay is replaced by the integer parameter. Otherwise, the assertion checks that when the register value changes, the output port value should match that value after the defined delay.

The last assertion example is for *CLEAR* function. The function causes the register content to be cleared when a defined port has a certain value [18]. In this assertion, the port is *CLEARPORT* and the register is *reg_CLEARPORTREG_s*. The assertion is shown in Figure 9.

```
// Clear function with port as condition
property CLEARFUNC_port_p;
  @(posedge ACLK) disable iff(!ARESETn)
    reg_function_tests_axi.CLEARPORT == 1'b1 &&
    !reg_function_tests_axi.write_CLEARPORTREG_active |=>
    !reg_function_tests_axi.reg_CLEARPORTREG_s;
endproperty
```

Figure 9: Assertion for clear function.

The assertion is evaluated on positive clock edges, when reset is inactive. This assertion requires that a register write operation does not happen at the same time. The register is implemented in RTL so that, if this condition is not included in the assertion, a simultaneous bus write could override the register content clear. This would cause the assertion to fail. It takes one clock cycle from the clear condition being true to register content being cleared, which is described with the nonoverlapping implication.

5.4 Results

Table 9 presents the results of this phase. Every SW interface function is included. Functions with and without assertion are annotated with different coloring. Also, wanted and unwanted results are marked with different color. Table 9 shows if the function had an assertion created, did it hold and were any additional constraints required. The table also tells if the function RTL implementation was fully covered with an assertion or not.

Table 9: Phase 1 results

	No assertion created for this function
	Failed/unwanted result
	Correct/wanted result

Function type	Syntax	Assertion created	Assertion holds	Constraint required	Assertion coverage
Input	NONE	no			
Input	PORT	no			

Function type	Syntax	Assertion created	Assertion holds	Constraint required	Assertion coverage
Input	PORT <name>	yes	yes	no	full
Input	PORT_MAX	no			
Input	PORT_MAX <name>	yes	yes	no	full
Input	LEVEL0	yes	yes	no	full
Input	LEVEL1	yes	yes	no	full
Input	LEVEL0 <enable_port_name>	no			
Input	LEVEL0 <enable_port_name>; PORT <port name>	yes	yes	no	full
Input	LEVEL1 <enable_port_name>	no			
Input	LEVEL1 <enable_port_name>; PORT <port name>	yes	yes	no	full
Input	LEVEL0 <register name>.<field name> ; PORT <port name>	yes	yes	no	full
Input	LEVEL1 <register name>.<field name> ; PORT <port name>	yes	yes	no	full
Input	EDGES0	yes	yes	no	full
Input	EDGES1	yes	yes	no	full
Input	EDGE01	no			
Input	EDGE10	no			
Input	EDGE01S0	yes	yes	no	full
Input	EDGE01S1	yes	yes	no	full
Input	EDGE10S0	yes	yes	no	full
Input	EDGE10S1	yes	yes	no	full
Input	REG OR <reg>.<field>	yes	yes	no	full
Input	REG_OP <register name> OR <register name>	yes	yes	no	full
Input	REG_OP <register name> AND <register name>	yes	yes	no	full
Input	REG_OP <register name> XOR <register name>	yes	yes	no	full
Input	REG_COND <register_name> <op> <value>	some cases	yes	no	full
Input	REG_COND <regis-ter_name>.<field_name> <op> <value>]	some cases	yes	no	full
Input	REG_OP_LOOP <OR|AND> <in-dex_var> <start_expr> <end_expr> <operand>	yes	yes	no	full
Input	PORT ; STROBE	no			
Input	PORT <port name>; STROBE <port name>	yes	yes	no	partial
Input	PORT ; SYNC=<n>	no			
Input	PORT <port name>; SYNC=<n>	yes	yes	no	full
Input	PORT ; SYNC=cnsyncax001	no			
Input	PORT ; SYNC=<name>	yes	yes	no	full
Input	PORT ; DELAY=<n>	no			
Input	PORT <port name>; DELAY=<n>	yes	yes	no	partial

Function type	Syntax	Assertion created	Assertion holds	Constraint required	Assertion coverage
Input	SET <0|1> WHEN <READ|WRITE> ACTIVE <name>	yes	yes	no	full
Output	NONE	no			
Output	PORT	no			
Output	PORT <name>	yes	yes	no	full
Output	PORT <name>(<number> downto <number>)	no			
Output	PORT; DELAY=<delay>	no			
Output	PORT <name>; DELAY=<delay>	yes	yes	no	partial
Output	PORT <name>(<number> downto <number>); DE-LAY=<delay>	no			
Output	PORT; CLOCK <name>; RESET <name>	no			
Output	PORT; CLOCK <name>; RESET <name>; SYNC=<name>	no			
Output	PORT <name>; CLOCK <name>; RESET <name>	no			
Output	PORT <name>; CLOCK <name>; RESET <name>; SYNC=<name>	yes	yes	no	full
Output	<op>	yes	yes	no	partial
Output	<op>_INPUT	yes	yes	no	full
Output	STROBE	yes	yes	no	partial
Output	PULSE; PORT	no			
Output	PULSE; PORT <port name>	yes	yes	no	full
Output	REG_PULSE	no			
Output	MUX <select> <output> <in-put> <input> ...	yes	yes	no	partial
Internal	ID	no			
Internal	<type>_<op>	no			
Register	<register name>	no			
Internal	WIRE	yes	yes	no	full
Internal	CONSTANT	yes	yes	no	full
Internal	BUS_ERROR	no			
Internal	POSTED_WRITE_ERROR	no			
Set	REG <register>.<field>=<value>	yes	yes	no	full
Set	REG <register>(<index>)=<value>	no			
Set	PORT <in-put_port_name>=<value>	yes	yes	no	full
Set	PORT <input_port_name>(<in-dex>)=<value>	yes	yes	no	partial
Clear	REG <register>.<field>=<value>	no			
Clear	REG <register>(<index>)=<value>	yes	yes	no	full

Function type	Syntax	Assertion created	Assertion holds	Constraint required	Assertion coverage
Clear	PORT <input_port_name>=<value>	yes	yes	no	full
Clear	PORT <input_port_name>(<index>)=<value>	yes	yes	no	partial
Enable	<REG|PORT> <expression>=<value>	yes	yes	no	full
Enable	<REG|PORT> <expression>=<value>;PARTIAL	no			
Misc	<UVM access type>	no			
Misc	<memory name>(<expr>)	no			
Misc	<memory name>(<expr>)(<data range>)	no			
	Wait Read <number>	no			
	Wait Read Port	no			
	Read Active Port	no			
	Write Active Port	no			

There were multiple reasons why a function did not have an assertion created for it. Majority of the functions that did not get an assertion could be covered by assertions created for other functions or at least with a very similar assertion. For example, if there was a function with two versions that were defined either with or without PORT <name> option, an assertion was created only for the one with the PORT <name>. This was acceptable as checking the most complex one would prove that the simpler ones could also be verified with an assertion. The same reason also applied for EDGE01 and EDGE10 functions as they were covered by the assertions of more complex EDGE functions.

One function had no assertion because it was not correctly implemented by the generator script. This function was REG_PULSE, which should create an internal signal to the SW interface that would show a one clock cycle pulse when the register is written. The register generated did not have this kind of functionality at all. This bug had gone unnoticed because the function was hardly used. No assertion was created for this function as plans to remove it completely from the SW interface emerged after this find.

A few function types were considered too difficult to verify with assertions. ID function was used to create a register to identify the SW interface. It contained constant parameters that should not be changed after the interface creation. It was seen unnecessary to write assertions for this function. Miscellaneous functions were for either mapping an external memory to the register address space or for providing register field access for universal verification methodology (UVM) test benches [21]. The memory related functions would have required an external memory to be connected to the interface. This meant that the assertions would have needed to cover also the memory blocks, which was out of scope

of this study. The UVM related function was already obsolete as other methods had replaced it, so no assertion for it was needed.

The assertions that were created performed well in register function verification. Every assertion proved to hold when a function was implemented correctly. None of the assertions required additional constraints as all needed conditions for related input and signal values could be included in the assertion itself. However, the code coverage of RTL implementation of registers was not satisfying. The coverage was a sum of RTL code being activated with FV tool RW assertions, which verified the bus, and the manually created function assertions. About one third of all register implementations were not fully covered by their assertions.

In this phase, assertions for most of the SW interface functionality added with register functions were created. However, some assertions did not fully cover the whole RTL implementation and functionality of the registers they were supposed to verify. In addition, the assertions did not have a uniform structure. Every assertion had been created individually without thinking of generalized structure. These problems meant that the assertions created in this phase were not good enough for the verification of a SW interface. However, further development of assertions was encouraged by the fact that they could perform well in interface function verification. The assertions should be enhanced so that they would completely cover the RTL implementation of SW interface registers. This could be achieved if the assertion could cover all possible ways to write, read or otherwise use the register value. The assertions should also have more general structure for easier understanding. A uniform structure would also make it easier to design a generation script for the assertions, which would automatize the assertion creation. These improvements were the specific tasks to be done in phase 2.

6. PHASE 2: ENHANCING PHASE 1 ASSERTIONS

Phase 2 of this thesis was about enhancing the assertions created in phase 1. Assertions had to cover RTL implementation of the register completely. They also needed to have as uniform structure as possible to ease assertion generation script development in the future. The idea of how this could be achieved was to integrate the assertions created into phase 1 to the register bus read and write assertions generated by the FV tool. By doing this, the register could be covered end-to-end with fewer assertions that would share a similar structure. This chapter describes the improved structure of the assertions with a couple of examples.

6.1 Structure of an enhanced assertion

The FV tool used in SW interface verification could generate assertions for the AXI4Lite bus *read* and *write* accesses to the interface registers. Figure 10 shows a generated *read* access assertion for a register called REG1.

```
REG1_BITS_access_reg_a: assert property (disable iff (!simple_regs.axi4lite_checker.ARESETn)
        AXI4Lite_read(12'h4) & 1'b1 & 1'b1 & simple_regs.RADDR_defined |->
        {AXI4Lite_handle_byte_enable((simple_regs.REG1_BITS [31:0]),
        simple_regs.RDATA,simple_regs.active_RSTRB)}[31:0] ==
        (\axi_if_0_mmap_simple_regs_REG1_BITS [31:0]) & (simple_regs.RRESP==2'b00)) else
        $error("REG1_BITS_access_reg_a: access of REG1_BITS does not work properly");
```

Figure 10: Generated assertion for AXI4Lite read operation.

In Figure 10, the assertion antecedent triggers when *read* operation is noticed to successfully complete on the AXI4Lite bus. After that, the read value, *RDATA* (retrieved with *AXI4Lite_handle_byte_enable function*) is compared to the register value stored by the FV tool. The *RDATA* and stored register value have to match and an *OKAY (2'b00)* response must be received from the DUV for the assertion to hold. Otherwise, an error is reported.

Figure 11 presents the *bus write* assertion generated for the same register. Here, the assertion checks that when a *bus write* operation happens, an *OKAY (2'b00)* response is received from the DUV.

```
REG1_write_reg_okay_a: assert property (disable iff (!simple_regs.ARESETn)
        AXI4Lite_write(12'h4) & 1'b1 |-> (simple_regs.BRESP==2'b00)) else
        $error("REG1_write_reg_okay_a: write of REG1_BITS causes a bus error");
```

Figure 11: Generated assertion for AXI4Lite write operation.

These autogenerated assertions of the FV tool are insufficient to verify SW interface functionality. Their purpose is to verify the bus protocol correctness which they do, but the SW interface functionality added with register functions is not verified. However, they could be improved to better or even completely verify the functionality of a register.

In the *bus write* assertion, it is not explicitly checked if the written data will make its way to the register. Due to this, the assertion does not provide complete code or functional coverage of the register write operation. The assertion would require a comparison between register data and *WDATA* to be added to the consequent to check that the data is really written into the register. This was tested with a couple of registers to see that it makes the assertion to fully cover the code and functionality of the register write operation. However, this modification was not implemented to all *bus write* assertions of every register because many of them had the *bus write* operation overridden by the register function functionality.

A function *get_comp_value*, which returns the actual register value of the DUV, a defined number of clock cycles earlier, was added to the *read* assertion. It replaced the variable originally holding the register value stored by the FV tool. This means that the *RDATA* was now compared with the register source value from an appropriate point of time some clock cycles before. The function takes the source name, which can be, e.g., a register, signal or port, of the input value and the amount of delay in clock cycles. The function is shown in Figure 12.

```
function automatic [31:0] get_comp_value(string name, int cycles);
        return $past(name, cycles);
endfunction
```

Figure 12: Function for bus read assertion.

For some SW interface functions, their functionality could be included in the FV tool assertions with the function presented in Figure 12. In this case, end-to-end coverage of register functionality was achieved by only modifying the *bus read* assertion. However, this was not possible for all SW interface functions. For those functions, only the register value was returned by the function and a separate assertion was created to verify the SW interface register function functionality. The separate assertion had a property defined on its own, which was then asserted to allow the reuse of the property, like in phase 1. These separate assertions were in most cases the same assertions already created in phase 1. The FV tool *bus write* assertion could also be modified to verify a couple of SW interface functions. However, a bus read assertion was still needed to achieve complete coverage of the register functionality. Examples of all different types of assertions are given in the next section.

6.2 Examples of enhanced assertions

The first enhanced assertion example is for the same register with an *input PORT* function as covered in section 5.3. The function adds a dedicated input port for the register [18]. The input port value is always written to the register as the RTL implementation for the register is such that the input port value always overwrites the value possibly coming through the bus. The assertion for this register is the FV tool *bus read* assertion with the

function *get_comp_value* described in Figure 12 added to it. The function returns the value of register input port from three clock cycles ago to be compared with the read register value, *RDATA*. The input port value is taken from three clock cycles ago because it is the time it takes for the value to be written into the register and then be visible in the *RDATA*, if it is immediately read by the bus. With this assertion, an end-to-end coverage of the register could be achieved. The assertion is shown in Figure 13.

```
BASICREG_BITS_access_reg_a: assert property (disable iff (!function_tests.ARESETn)
        AXI4Lite_read(12'h4) & 1'b1 & 1'b1 & function_tests.RADDR_defined |->
        {AXI4Lite_handle_byte_enable((function_tests.BASICREG_BITS [31:0]),
        function_tests.RDATA,function_tests.active_RSTRB)}[31:0] ==
        get_comp_value(function_tests.BASIC_REG_IN, 3) & (function_tests.RRESP==2'b00)) else
        $error("BASICREG_BITS_access_reg_a: access of BASICREG_BITS does not work properly");
```

Figure 13: Enhanced assertion for input PORT function.

The second example is for the same register with *output DELAY* function as presented in section 5.3. A dedicated output port is created that shows the register value with a defined delay [18]. This functionality could not easily be integrated into the *bus read* assertion because of the separate output port. Only the *get_comp_value* function was included in read assertion to return a register value from two clock cycles ago to the comparison. The delay is one clock cycle less than in the first example, because the retrieved value was not the input port value as in the first example. The modified read assertion is shown in Figure 14.

```
DELAYREG2_BITS_access_reg_a: assert property (disable iff (!function_tests.ARESETn)
        AXI4Lite_read(12'hc) & 1'b1 & 1'b1 & function_tests.RADDR_defined |->
        {AXI4Lite_handle_byte_enable((function_tests_DELAYREG2_BITS [3:0]),
        function_tests.RDATA,function_tests.active_RSTRB)}[3:0] ==
        get_comp_value(function_tests.reg_DELAYREG2_s, 2) & (function_tests.RRESP==2'b00)) else
        $error("DELAYREG2_BITS_access_reg_a: access of DELAYREG2_BITS does not work properly");
```

Figure 14: Enhanced assertion for output DELAY function.

Since the *bus read* assertion could not be modified to verify *DELAY* function functionality, it had to be done in a separate assertion. The assertion is almost the same as the one created in phase 1 for this register. It checks that when the output port value changes, the new output port value equals the register value defined a given number of clock cycles ago. The number of clock cycles was defined in the *DELAY* function definition. This precise definition of delay in the assertion differs from the parametrized delay of the assertion in phase 1. This approach was chosen because it makes it less complicated and a new assertion would anyway be required for every new register. The separate assertion property is presented in Figure 15.

```
property DELAYFUNC_p;
 disable iff(!function_tests.axi4lite_checker.ARESETn)
     $changed(function_tests.DELAY_REG2_OUT) |->
 function_tests.DELAY_REG2_OUT == $past(function_tests.reg_DELAYREG2_s, 3);
endproperty
```

Figure 15: Output DELAY function property.

The third example is for the same register with *CLEAR* function as in section 5.3. It adds an input port that can be used for clearing register content by writing a certain value to the port [18]. This functionality was not feasible to be integrated into the *bus write* or *read* assertion. However, a similar approach as for the register with a *DELAY* function could be used for this register function. The *get_comp_value* function was added to *bus read* assertion and the phase 1 assertion was used to separately verify the *CLEAR* function functionality to achieve complete coverage of the register.

Some registers with added functionality could be verified by adding a different function than *get_comp_value* to the *bus read* assertion. An example of this is a register with *REG_OP* function. This function causes the register value to be the result of *AND, OR* or *XOR* operation of two different register contents. The result is always stored in the register as it overwrites the *bus write* operation. The register could therefore be completely verified by adding a function that returns the computed to the comparison with *RDATA* in the *bus read* assertion. The assertion for a register with this functionality is shown in Figure 16.

```
INPUTREGOPXORREG_BITS_access_reg_a: assert property (disable iff (!function_tests.ARESETn)
        AXI4Lite_read(12'h128) & 1'b1 & 1'b1 & function_tests.RADDR_defined |->
       {AXI4Lite_handle_byte_enable((function_tests_INPUTREGOPXORREG_BITS [15:0]),
       function_tests.RDATA,function_tests.active_RSTRB)}[15:0] ==
       (i_regoxor_val) & (function_tests.RRESP==2'b00)) else
       $error("INPUTREGOPXORREG_BITS_access_reg_a: access of INPUTREGOPXORREG_BITS does not work properly");

function [15:0] i_regoxor_val();
       return $past(function_tests.reg_INPUTMAXREG_s ^ function_tests.reg_INPUTDELAYREG_s, 3);
endfunction
```

Figure 16: Enhanced input REG_OP function assertion.

Few register functions could be verified by modifying the *bus write* assertion. *Input DELAY* function adds a delay to the *bus write* operation [18]. The written value is visible in the register after a defined delay. The delay is measured in clock cycles. The *bus write* assertion was modified to include an antecedent condition of *WSTRB* signal being '1' for the defined delay number of clock cycles before allowing the register value to be changed (refer to [20] for further details). When a *bus write* occurred and *WSTRB* allowed register content to be changed, the assertion checked that write response was *OK (2'b00)* and that the register value was the *WDATA* value from the defined delay amount of cycles before. The assertion is shown in Figure 17.

```
DELAYREG1_write_reg_okay_a: assert property (disable iff (!function_tests.ARESETn)
        AXI4Lite_write(12'h8) & 1'b1 & ($past(function_tests.WSTRB[0:0], 3) == 1'b1) |->
        (function_tests.BRESP==2'b00) &&
        (function_tests.reg_DELAYREG1_s == $past(function_tests.WDATA[3:0], 3))) else
        $error("DELAYREG1_write_reg_okay_a: write of DELAYREG1_BITS causes a bus error");
```

Figure 17: Enhanced input DELAY function assertion.

7. RESULTS OF SW INTERFACE VERIFICATION

This chapter presents the combined results of the two phases of assertion-based FV done on the SW interface functionality. The results cover all SW interface functionality that was verified with the methods described in section 5.2.2. Comparison between the two phases and the metrics that present how well the set goals were met are also included.

Section 5.4 already described the results of phase 1 in detail and explained, why some register functions were not separately verified. The functions that were not verified could either be verified with an assertion created for other, more complex version of the same function. Also, some of the functions were found out to be bugged or redundant before an assertion was created, or the function was not simply feasible to verify with an assertion, which is why no assertion was created for them. These functions were not verified in phase 2 either and were therefore excluded from the results table.

Table 10 shows the results of register functionality verification from phase 1 and 2. It tells if the assertion(s) created held on a register with correct register function implementation and fired when the implementation was wrong. Also, the need for possible additional constraints is shown. Assertion coverage column describes if the assertion(s) covered the functionality and code of the register with a function completely or not. Multiple assertions column shows if more assertions than only the SW interface bus read and write assertions were required for the whole register verification. The color coding used in the table is presented above it.

Table 10: Verification results

Failed/unwanted result
Correct/wanted result

		Phase 1				Phase 2			
Function type	**Syntax**	**Assertion holds**	**Constraint required**	**Assertion coverage**	**Multiple assertions**	**Assertion holds**	**Constraint required**	**Assertion coverage**	**Multiple assertions**
Input	PORT <name>	yes	no	full	yes	yes	no	full	no
Input	PORT_MAX <name>	yes	no	full	yes	yes	no	full	yes
Input	LEVEL0	yes	no	full	yes	yes	no	full	yes
Input	LEVEL1	yes	no	full	yes	yes	no	full	yes
Input	LEVEL0 <enable_port_name>; PORT <port name>	yes	no	full	yes	yes	no	full	yes
Input	LEVEL1 <enable_port_name>; PORT <port name>	yes	no	full	yes	yes	no	full	yes
Input	LEVEL0 <register name>.<field name> ; PORT <port name>	yes	no	full	yes	yes	no	full	yes
Input	LEVEL1 <register name>.<field name> ; PORT <port name>	yes	no	full	yes	yes	no	full	yes
Input	EDGES0	yes	no	full	yes	yes	no	full	yes
Input	EDGES1	yes	no	full	yes	yes	no	full	yes
Input	EDGE01S0	yes	no	full	yes	yes	no	full	yes
Input	EDGE01S1	yes	no	full	yes	yes	no	full	yes
Input	EDGE10S0	yes	no	full	yes	yes	no	full	yes
Input	EDGE10S1	yes	no	full	yes	yes	no	full	yes
Input	REG OR <reg>.<field>	yes	no	full	yes	yes	no	full	no

		Phase 1				Phase 2			
Function type	**Syntax**	**Assertion holds**	**Constraint required**	**Assertion coverage**	**Multiple assertions**	**Assertion holds**	**Constraint required**	**Assertion coverage**	**Multiple assertions**
Input	REG_OP <register name> OR <register name>	yes	no	full	yes	yes	no	full	no
Input	REG_OP <register name> AND <register name>	yes	no	full	yes	yes	no	full	no
Input	REG_OP <register name> XOR <register name>	yes	no	full	yes	yes	no	full	no
Input	REG_COND <register_name> <op> <value>	yes	no	full	yes	yes	no	full	yes
Input	REG_COND <register_name>.<field_name> <op> <value>]	yes	no	full	yes	yes	no	full	yes
Input	REG_OP_LOOP <OR|AND> <index_var> <start_expr> <end_expr> <operand>	yes	no	full	yes	yes	no	full	no
Input	PORT <port name>; STROBE <port name>	yes	no	partial	yes	yes	no	full	yes
Input	PORT <port name>; SYNC=<n>	yes	no	full	yes	yes	no	full	no
Input	PORT ; SYNC=<name>	yes	no	full	yes	yes	no	full	no
Input	PORT <port name>; DELAY=<n>	yes	no	partial	yes	yes	no	full	no
Input	SET <0|1> WHEN <READ|WRITE> ACTIVE <name>	yes	no	full	yes	yes	no	full	yes
Output	PORT <name>	yes	no	full	yes	yes	no	full	yes
Output	PORT <name>; DELAY=<delay>	yes	no	partial	yes	yes	no	full	yes
Output	PORT <name>; CLOCK <name>; RESET <name>; SYNC=<name>	yes	no	full	yes	yes	no	full	yes
Output	<op>	yes	no	partial	yes	yes	no	full	yes

		Phase 1				Phase 2			
Function type	**Syntax**	**Assertion holds**	**Constraint required**	**Assertion coverage**	**Multiple assertions**	**Assertion holds**	**Constraint required**	**Assertion coverage**	**Multiple assertions**
Output	<op>_INPUT	yes	no	full	yes	yes	no	full	yes
Output	STROBE	yes	no	partial	yes	yes	no	full	yes
Output	PULSE; PORT <port name>	yes	no	full	yes	yes	no	full	yes
Output	MUX <select> <output> <input> <input> ...	yes	no	partial	yes	yes	no	full	yes
Internal	WIRE	yes	no	full	yes	yes	no	full	no
Internal	CONSTANT	yes	no	full	yes	yes	no	full	yes
Set	REG <register>.<field>=<value>	yes	no	full	yes	yes	no	full	yes
Set	PORT <input_port_name>=<value>	yes	no	full	yes	yes	no	full	yes
Set	PORT <input_port_name>(<index>)=<value>	yes	no	partial	yes	yes	no	full	yes
Clear	REG <register>(<index>)=<value>	yes	no	full	yes	yes	no	full	yes
Clear	PORT <input_port_name>=<value>	yes	no	full	yes	yes	no	full	yes
Clear	PORT <input_port_name>(<index>)=<value>	yes	no	partial	yes	yes	no	full	yes
Enable	<REG|PORT> <expression>=<value>	yes	no	full	yes	yes	no	full	yes

The first thing to note from the results is that in both phases assertions to verify correct implementations of registers with a register function could be created. Phase 1 assertions could verify correct implementation of the functionality added with register functions. This meant that the non-working register functions were already discovered in phase 1. However, there was only a function REG_PULSE that was bugged.

The second thing to note is that the assertions were created and checked without any additional constraints to block certain design behavior. The included constraints were only to ensure that the FV tool used valid bus protocol behavior and input data. The absence of overconstraining was desired because restricting some otherwise valid input behavior to allow a certain assertion to work would prevent some other functionality of the SW interface from occurring. Overconstraining was not needed because the required conditions could be included in the created assertions.

The differences between phase 1 and 2 results come from the functional and code coverage of register functionality and required amount of assertions to achieve the coverage. In phase 1, roughly one third of the possible register functionality could not be completely verified with the bus RW and the register function assertions. In phase 2, when these two assertion types were integrated as fully as possible into each other, all relevant register functionality could be verified.

The only functionalities that were not covered with phase 2 assertions were the reset value setting of most registers and the *bus write* operation of some registers. The reset value setting was not checked because all created assertions were disabled during reset. However, proper usage of the default register values was covered immediately after reset by the assertions. In addition, the reset values were correct for all defined registers so the register value setting is working as intended. The *bus write* operation was not covered for those registers that it was redundant. Figure 18 presents the RTL code for a register with a dedicated input port. Assertion created for that register was presented in section 6.2. It provided end-to-end coverage of the register by checking in *bus read* assertion that the input port value is always seen after a proper delay in the output port. In this figure, it can be seen that the reset value setting is not covered and that the *bus write* is not covered due to it being redundant functionality and therefore, not verified by the assertion for this register.

```
write_BASICREG_p : process (ARESETn, ACLK)
begin
  if ARESETn = '0' then
    -- Field : BITS
    reg_BASICREG_s(31 downto 0) <= "00000000000000000000000000000000";
  elsif ACLK'event and ACLK = '1' then
    -- Register BASICREG field BITS.
    -- Access : read-write (RW)
    -- Input Function : 'PORT BASIC_REG_IN'
    -- Output Function : 'PORT BASIC_REG_OUT'
    -- Read Action : 'none'
    -- Write Action : 'none'
    if write_BASICREG_active then
      for b in 31 downto 0 loop
        if bit_selected(write_select_byte, b) then
          reg_BASICREG_s(b) <= write_data(b);
        end if;
      end loop;
    end if;
    for b in 31 downto 0 loop
      reg_BASICREG_s(b) <= BASIC_REG_IN(b);
    end loop;
  end if;
end process write_BASICREG_p;
```

Figure 18: Quantify code coverage snapshot.

Structural Coverage Overview

Status		Statements		Branches	
1	covered	490	59.54%	305	61.37%
R	reached	0	0.00%	0	0.00%
U	unknown	0	0.00%	1	0.20%
0R	unobserved	318	38.64%	190	38.23%
0	uncovered	15	1.82%	1	0.20%
0C	constrained	0	0.00%	0	0.00%
0D	dead	0	0.00%	0	0.00%
Sum	**quantify targets**	**823**		**497**	

Figure 19: Quantify structural coverage overview.

The coverage was assessed from the FV tool Quantify reports combined with the information if the assertion could verify the correct behavior based on the witness and counterexample reports of the FV tool. A Quantify structural coverage report of a SW interface with all phase two assertions included is presented in Figure 19. The report shows that roughly 60% of the design structures were exhaustively covered by the assertions. The reason why the number is so low is because of the *bus write* operations and register reset value setting. As noted in section 6.1, the *FV tool bus write* assertions did not check that the written value really ended up to the register. A simple added comparison would have improved the assertion to cover this. It was tested on few registers that the improved bus write assertion can cover the register write completely. These improvements were left out from the assertion checking due to them being similar for any register and would have just taken development and verification time to add them for every register. In addition, the *bus write* operation was redundant for many registers due to being overridden by the

functionality added with a register function. For these reasons, the *bus write* operation code is not observed by any assertion for majority of the registers. This and the fact that reset value setting was not checked caused the total coverage to be relatively low. However, the coverage of the relevant register code and functionality was 100%. A more detailed report of the coverage of individual assertions is included in Appendix A. The report in the appendix also shows some not proven assertions that were created by the FV tool. These assertions checked bus features not present in the design, which caused them to be inconclusive.

Phase 1 assertions for register functions were tried to be integrated into *FV tool bus RW* assertions to reduce the number of assertions needed and to have a more uniform structure with the different assertions. As seen from Table 10, only about one third of the registers with functions could be completely verified with only modified *bus RW* assertions. Many functions added an additional port or a method to change, read or use the register value that was not compatible enough with the *bus RW* assertions to be integrated into them. Still, a lower total number of assertions with more general structure was achieved in phase 2, which is better for potential assertion generation script development.

One difference between phase 1 and 2 not visible in Table 10 was the increased runtime of assertion checking in phase 2 compared to phase 1. The runtime of a single assertion check was in the worst cases increased from few seconds to some tens of seconds. Also, the Quantify runtimes increased significantly. This was caused by the fact that the phase 2 assertions were more complex than the assertions in phase 1. The runtimes are not a problem for a smaller scale SW interface like the ones used in this thesis. However, for larger SW interfaces with hundreds or even thousands of registers that can have multiple functions defined to them, the runtimes would further increase and could become an issue. It is not feasible to run assertion checks that last for tens of hours if it needs to be done multiple times for a single SW interface. However, the checks would be usually run, when the SW interface changes, which should not happen often in a single project. In that case, the longer runtime of assertion checks does not matter that much. For larger SW interfaces, assertion checks could be run with a limited set of assertions and/or with a lower effort level to reduce the checking time.

The verified SW interface had only registers with a single function. It is allowed to define multiple functions for a single register. The registers with multiple functions could be verified by combining the assertions created for individual register functions. These multiple function cases were out of the scope of this thesis but would be one of the first steps to be considered if an assertion generation script was created.

8. CONCLUSIONS

The goal of this thesis was to develop a new assertion-based formal verification method to verify SoC module SW interfaces used in Nokia. The developed method was meant to support or replace the former simulation-based verification methods. This required modelling the design intent by creating assertions and then using FV tool assertion checks to verify the SW interface RTL implementation. From the results, suggestions could be made if and on what scale a new assertion-based FV platform could be utilized in SW interface verification.

The assertion creation and SW interface verification were done in two phases. In the first phase, assertions for the SW interface register functions were created. These assertions were used with the bus RW assertions created by the FV tool to verify if SW interfaces were generated correctly. The assertion-based FV performed well in SW interface verification as some bugs were found and the functionality added with register functions could be fully verified with the created assertions. However, the coverage of the design code was not good enough, and the assertions did not have a uniform structure. This meant that improvements were needed.

In the second phase, the assertions from first phase were enhanced by integrating them as well as possible into the FV tool bus RW assertions. This provided a better coverage of the design code and functionality with fewer and more similarly created assertions. Still, many register functions required a separate assertion in addition to the bus RW assertions. However, nearly complete code and functional coverage of the SW interface was achieved with the new assertions with tolerable verification runtimes. This is better than what the current simulation-based methods can provide.

The developed assertion-based FV platform could provide a new way to verify SW interfaces. A full coverage of the design could be achieved with acceptable runtimes. However, a relatively simple SW interface was used in the experiments. A larger and more complex SW interface would take much more effort and time to be verified with this approach. Also, a generation script for the assertions would be required to make their creation easier since manually defining them takes a lot of time. Therefore, the platform in its current state suits best for smaller or for partial verification of larger SW interfaces.

Still, it can be said that this thesis achieved its goals. Assertions verifying SW interface functionality were created. The SW interface generation flow was verified and the several found bugs were fixed so it is in a better condition now than before this thesis. A platform to develop an assertion generation script to enable automated additional verification method for SW interfaces was produced. The developed platform should in its current state be considered as an additional way to verify small or parts of bigger SW interfaces.

9. REFERENCES

[1] R. Djemal, M.A. Dhouib, S. Dellacherie, R. Tourki, A novel formal verification approach for RTL hardware IP cores, Computer Standards & Interfaces, Vol. 27, Iss. 6, 2005, pp. 637-651.

[2] M. Tallgren, Case studies on formal assertion-based verification of a SoC design, Master's Thesis, University of Oulu, Faculty of Information Technology and Electrical Engineering, Oulu, 2017.

[3] A. Meyer, Principles of functional verification, Newnes Newton, MA USA, 2004.

[4] B.J. LaMeres, VHDL (Part 1), Introduction to Logic Circuits & Logic Design with VHDL, Springer International Publishing, Switzerland, 2017, pp. 139-174.

[5] J. Wang, J. Shao, Y. Li, J. Ding, Survey on Formal Verification Methods for Digital IC, 2009 Fourth International Conference on Internet Computing for Science and Engineering, Harbin, China, 2009, pp. 164-168.

[6] IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language, IEEE, IEEE STD 1800-2017, 2018.

[7] E. Cerny, S. Dudani, J. Havlicek, D. Korchemny, SVA: The Power of Assertions in SystemVerilog, Second Edition ed., Springer International Publishing, Switzerland, 2015.

[8] A.B. Mehta, ASIC/SoC Functional Design Verification: A Comprehensive Guide to Technologies and Methodologies, Springer International Publishing, Switzerland, 2017.

[9] A.B. Mehta, SystemVerilog assertions and functional coverage: guide to language, methodology and applications, Springer International Publishing, Switzerland, 2016.

[10] Reducing Verification Risk with Formal-Based Observation Coverage, Onespin Solutions GmbH, 2014, Available (accessed 2.2.2018) https://www.onespin.com/resources/white-papers/.

[11] J. Aynsley, Become an SVA Expert in One Hour, Doulos Webinars (web-based seminar), January 2018.

[12] Questa Formal Verification, Mentor, A Siemens Business, web page, Available (accessed 26.2.2018): https://www.mentor.com/products/fv/questa-formal/.

[13] VC formal: Next-Generation Formal Verification, Synopsys Inc., web page, Available (accessed 26.2.2018) https://www.synopsys.com/verification/static-and-formal-verification/vc-formal.html.

[14] 360 DV-verify: Unified, coverage-driven assertion-based verification, including a full automated apps library, Onespin Solutions GmbH, web page, Available (accessed 14.4.2018) https://www.onespin.com/products/360-dv-verify/.

[15] IEEE/IEC International Standard - IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows, IEEE, IEC 62014-4 IEEE Std 1685-2009, 2015.

[16] A.B. Mehta, SystemVerilog Assertions and Functional Coverage, Springer International Publishing, Switzerland, 2016.

[17] User Manual: Onespin 360™ DV Design Verification Solution, Onespin Solutions GmbH, 2017, Copyright © 2005-2017 Onespin Solutions GmbH.

[18] J. Nousiainen, Register Generator User Guide, Nokia Mobile Networks, 2017, unpublished, Copyright © Nokia Networks 2018.

[19] ABOUT IP-XACT (IEEE 1685): IEEE 1685 Compliance Lab, Magilllem Design Services, web page, Available (accessed 20.3.2018) http://www.magillem.com/ip-xact-ieee-1685/.

[20] AMBA AXI and ACE Protocol Specification AXI3, AXI4, and AXI4-Lite, ACE and ACE-Lite, ARM limited, 2013, Available (accessed 20.4.2018) http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0022e/index.html, Copyright © ARM Limited 2013.

[21] Accellera Universal Verification Methodology (UVM) 1.2 User Guide, Accellera Systems Iniative, Available (accessed 19.4.2018) http://accellera.org/downloads/standards/uvm.

APPENDIX A: QUANTIFY ASSERTION COVERAGE REPORT

Assertion Coverage

Id	Property	Kind	Proof Result	Proof Radius	Cover Result	Cover Radius	Quantified
0	sva/axi4lite_checker/axi_checker/parameter_DATA_SIZE_legal_a	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
1	sva/axi4lite_checker/axi_checker/parameter_MAX_PEND_READS_sufficient_a	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
2	sva/axi4lite_checker/axi_checker/parameter_MAX_PEND_WRITES_sufficient_a	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
3	sva/axi4lite_checker/axi_checker/slave/B_legal_exclusive_burst_a	assert	FORMAL_PROOF	infinite	COVER_VACUOUS	infinite	no
4	sva/axi4lite_checker/axi_checker/slave/B_no_access_after_reset_a	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
5	sva/axi4lite_checker/axi_checker/slave/B_no_unrequested_response_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
6	sva/axi4lite_checker/axi_checker/slave/B_stable_signals_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
7	sva/axi4lite_checker/axi_checker/slave/RLAST_correct_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
8	sva/axi4lite_checker/axi_checker/slave/R_legal_exclusive_burst_a	assert	FORMAL_PROOF	infinite	COVER_VACUOUS	infinite	no
9	sva/axi4lite_checker/axi_checker/slave/R_no_access_after_reset_a	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
10	sva/axi4lite_checker/axi_checker/slave/R_no_unrequested_data_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
11	sva/axi4lite_checker/axi_checker/slave/R_stable_signals_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
12	sva/register_check/axi_if_0_mmap_reg_function_tests_BASICREG_BITS_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	5	yes
13	sva/register_check/axi_if_0_mmap_reg_function_tests_BASICREG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
14	sva/register_check/axi_if_0_mmap_reg_function_tests_BUSERRORREG_BITS_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
15	sva/register_check/axi_if_0_mmap_reg_function_tests_BUSERRORREG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
16	sva/register_check/axi_if_0_mmap_reg_function_tests_CLEARPORTINDEXREG_BITS_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
17	sva/register_check/axi_if_0_mmap_reg_function_tests_CLEARPORTINDEXREG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
18	sva/register_check/axi_if_0_mmap_reg_function_tests_CLEARPORTREG_BITS_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
19	sva/register_check/axi_if_0_mmap_reg_function_tests_CLEARPORTREG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
20	sva/register_check/axi_if_0_mmap_reg_function_tests_CLEARREG_BIT_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
21	sva/register_check/axi_if_0_mmap_reg_function_tests_CLEARREG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
22	sva/register_check/axi_if_0_mmap_reg_function_tests_CLOCKSYNCREG_BITS_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
23	sva/register_check/axi_if_0_mmap_reg_function_tests_CLOCKSYNCREG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
24	sva/register_check/axi_if_0_mmap_reg_function_tests_CONSTANTREG_BIT_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
25	sva/register_check/axi_if_0_mmap_reg_function_tests_DELAYREG1_BITS_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
26	sva/register_check/axi_if_0_mmap_reg_function_tests_DELAYREG1_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
27	sva/register_check/axi_if_0_mmap_reg_function_tests_DELAYREG2_BITS_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	5	yes
28	sva/register_check/axi_if_0_mmap_reg_function_tests_DELAYREG2_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
29	sva/register_check/axi_if_0_mmap_reg_function_tests_EDGE01S0REG_BITS_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
30	sva/register_check/axi_if_0_mmap_reg_function_tests_EDGE01S0REG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
31	sva/register_check/axi_if_0_mmap_reg_function_tests_EDGE01S1REG_BIT_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
32	sva/register_check/axi_if_0_mmap_reg_function_tests_EDGE01S1REG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
33	sva/register_check/axi_if_0_mmap_reg_function_tests_EDGE10S0REG_BIT_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
34	sva/register_check/axi_if_0_mmap_reg_function_tests_EDGE10S0REG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
35	sva/register_check/axi_if_0_mmap_reg_function_tests_EDGE10S1REG_BITS_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
36	sva/register_check/axi_if_0_mmap_reg_function_tests_EDGE10S1REG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
37	sva/register_check/axi_if_0_mmap_reg_function_tests_EDGES0REG_BIT_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
38	sva/register_check/axi_if_0_mmap_reg_function_tests_EDGES0REG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
39	sva/register_check/axi_if_0_mmap_reg_function_tests_EDGES1REG_BIT_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
40	sva/register_check/axi_if_0_mmap_reg_function_tests_EDGES1REG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes

41	sva/register_check/axi_if_0_mmap_reg_function_tests_ENABLEPORTREG_BIT_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
42	sva/register_check/axi_if_0_mmap_reg_function_tests_ENABLEPORTREG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
43	sva/register_check/axi_if_0_mmap_reg_function_tests_ENABLEREGREG_BIT_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
44	sva/register_check/axi_if_0_mmap_reg_function_tests_ENABLEREGREG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
45	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTCONDFIELDREG_DATA_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
46	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTCONDREG_BITS_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
47	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTDELAYREG_BITS_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
48	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTDELAYREG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
49	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTLEVEL0_BITS_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
50	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTLEVEL0_ENABLE_BITS_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
51	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTLEVEL0_ENABLE_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
52	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTLEVEL0_REGENABLE_BITS_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
53	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTLEVEL0_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
54	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTLEVEL1_BITS_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
55	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTLEVEL1_ENABLE_BITS_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
56	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTLEVEL1_ENABLE_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
57	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTLEVEL1_REGENABLE_BITS_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
58	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTLEVEL1_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
59	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTMAXREG_DATA_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
60	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTMAXREG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
61	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTNAMEDSYNCREG_BITS_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
62	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTNAMEDSYNCREG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
63	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTORREG_BIT_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	5	yes
64	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTORREG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
65	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTREGOPANDREG_BITS_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	5	yes
66	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTREGOPANDREG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
67	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTREGOPLOOPANDREG_BIT_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	5	yes
68	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTREGOPLOOPANDREG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
69	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTREGOPLOOPORREG_BIT_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	5	yes
70	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTREGOPLOOPORREG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
71	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTREGOPORREG_BITS_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	5	yes
72	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTREGOPORREG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
73	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTREGOPXORREG_BITS_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	5	yes
74	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTREGOPXORREG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
75	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTSETWHENREG_BIT0_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
76	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTSETWHENREG_BIT1_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
77	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTSETWHENREG_BIT2_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
78	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTSETWHENREG_BITS3_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
79	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTSTROBEREG_BITS_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	5	yes
80	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTSTROBEREG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
81	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTSYNCREG_BITS_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	5	yes

82	sva/register_check/axi_if_0_mmap_reg_function_tests_INPUTSYNCREG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
83	sva/register_check/axi_if_0_mmap_reg_function_tests_MULTIDIMREG_0_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
84	sva/register_check/axi_if_0_mmap_reg_function_tests_MULTIDIMREG_1_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
85	sva/register_check/axi_if_0_mmap_reg_function_tests_MULTIDIMREG_2_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
86	sva/register_check/axi_if_0_mmap_reg_function_tests_MULTIDIMREG_DATA_0_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
87	sva/register_check/axi_if_0_mmap_reg_function_tests_MULTIDIMREG_DATA_1_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
88	sva/register_check/axi_if_0_mmap_reg_function_tests_MULTIDIMREG_DATA_2_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
89	sva/register_check/axi_if_0_mmap_reg_function_tests_NBITREG_HIGHBITS_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	5	yes
90	sva/register_check/axi_if_0_mmap_reg_function_tests_NBITREG_LOWBITS_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
91	sva/register_check/axi_if_0_mmap_reg_function_tests_NBITREG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
92	sva/register_check/axi_if_0_mmap_reg_function_tests_OPERATIONANDREG_BITS_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
93	sva/register_check/axi_if_0_mmap_reg_function_tests_OPERATIONANDREG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
94	sva/register_check/axi_if_0_mmap_reg_function_tests_OPERATIONINPUTANDREG_DATA_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
95	sva/register_check/axi_if_0_mmap_reg_function_tests_OPERATIONINPUTANDREG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
96	sva/register_check/axi_if_0_mmap_reg_function_tests_OPERATIONINPUTORREG_DATA_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
97	sva/register_check/axi_if_0_mmap_reg_function_tests_OPERATIONINPUTORREG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
98	sva/register_check/axi_if_0_mmap_reg_function_tests_OPERATIONORREG_DATA_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
99	sva/register_check/axi_if_0_mmap_reg_function_tests_OPERATIONORREG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
100	sva/register_check/axi_if_0_mmap_reg_function_tests_OUTPUTMUXREG_BIT1_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
101	sva/register_check/axi_if_0_mmap_reg_function_tests_OUTPUTMUXREG_BIT2_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
102	sva/register_check/axi_if_0_mmap_reg_function_tests_OUTPUTMUXREG_BIT3_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
103	sva/register_check/axi_if_0_mmap_reg_function_tests_OUTPUTMUXREG_MODE_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
104	sva/register_check/axi_if_0_mmap_reg_function_tests_OUTPUTMUXREG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
105	sva/register_check/axi_if_0_mmap_reg_function_tests_PULSEREG_DATA_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
106	sva/register_check/axi_if_0_mmap_reg_function_tests_PULSEREG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
107	sva/register_check/axi_if_0_mmap_reg_function_tests_REGPULSEREG_DATA_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
108	sva/register_check/axi_if_0_mmap_reg_function_tests_REGPULSEREG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
109	sva/register_check/axi_if_0_mmap_reg_function_tests_SETPORTINDEXREG_BIT_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
110	sva/register_check/axi_if_0_mmap_reg_function_tests_SETPORTINDEXREG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
111	sva/register_check/axi_if_0_mmap_reg_function_tests_SETPORTREG_BIT_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
112	sva/register_check/axi_if_0_mmap_reg_function_tests_SETPORTREG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
113	sva/register_check/axi_if_0_mmap_reg_function_tests_SETREG_BITS_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
114	sva/register_check/axi_if_0_mmap_reg_function_tests_SETREG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
115	sva/register_check/axi_if_0_mmap_reg_function_tests_SINGLEBITREG_BIT_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	5	yes
116	sva/register_check/axi_if_0_mmap_reg_function_tests_SINGLEBITREG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
117	sva/register_check/axi_if_0_mmap_reg_function_tests_STROBEREG_DATA_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
118	sva/register_check/axi_if_0_mmap_reg_function_tests_STROBEREG_write_reg_okay_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
119	sva/register_check/axi_if_0_mmap_reg_function_tests_WIREREG_BIT_access_reg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
120	sva/register_check/my_BASICOUTPUT_a	assert	FORMAL_PROOF	infinite	COVER_PASS	2	yes
121	sva/register_check/my_CLEARPORTINDEX_a	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
122	sva/register_check/my_CLEARPORT_a	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes

123	sva/register_check/my_CLEARREG_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
124	sva/register_check/my_CLOCKSYNC_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
125	sva/register_check/my_CONSTANT_a	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
126	sva/register_check/my_EDGE01S0_a[0]/register_checker_m_assert_1	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
127	sva/register_check/my_EDGE01S0_a[1]/register_checker_m_assert_1	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
128	sva/register_check/my_EDGE01S0_a[2]/register_checker_m_assert_1	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
129	sva/register_check/my_EDGE01S0_a[3]/register_checker_m_assert_1	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
130	sva/register_check/my_EDGE01S1_a	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
131	sva/register_check/my_EDGE10S0_a	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
132	sva/register_check/my_EDGE10S1_a[0]/register_checker_m_assert_0	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
133	sva/register_check/my_EDGE10S1_a[1]/register_checker_m_assert_0	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
134	sva/register_check/my_EDGE10S1_a[2]/register_checker_m_assert_0	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
135	sva/register_check/my_EDGE10S1_a[3]/register_checker_m_assert_0	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
136	sva/register_check/my_EDGES0_a	assert	FORMAL_PROOF	infinite	COVER_PASS	2	yes
137	sva/register_check/my_EDGES1_a	assert	FORMAL_PROOF	infinite	COVER_PASS	2	yes
138	sva/register_check/my_ENABLEPORT_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
139	sva/register_check/my_ENABLEREG_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
140	sva/register_check/my_INPUTCONDREG_a	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
141	sva/register_check/my_INPUTCOND_a	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
142	sva/register_check/my_INPUTLEVEL0_a	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
143	sva/register_check/my_INPUTLEVEL0_enaport_a	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
144	sva/register_check/my_INPUTLEVEL0_enareg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
145	sva/register_check/my_INPUTLEVEL1_a	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
146	sva/register_check/my_INPUTLEVEL1_enaport_a	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
147	sva/register_check/my_INPUTLEVEL1_enareg_a	assert	FORMAL_PROOF	infinite	COVER_PASS	2	yes
148	sva/register_check/my_INPUTMAX_a	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
149	sva/register_check/my_INPUTNAMEDSYNC_a	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
150	sva/register_check/my_INPUTSET0WHEN_READ_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
151	sva/register_check/my_INPUTSET0WHEN_WRITE_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
152	sva/register_check/my_INPUTSET1WHEN_READ_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
153	sva/register_check/my_INPUTSET1WHEN_WRITE_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
154	sva/register_check/my_INPUTSTROBE_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
155	sva/register_check/my_OPERATIONAND_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
156	sva/register_check/my_OPERATIONINPUTAND_a	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
157	sva/register_check/my_OPERATIONINPUTOR_a	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
158	sva/register_check/my_OPERATIONOR_a	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
159	sva/register_check/my_OUTMUX1_00_a	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
160	sva/register_check/my_OUTMUX1_01_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
161	sva/register_check/my_OUTMUX1_10_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
162	sva/register_check/my_OUTMUX1_other_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
163	sva/register_check/my_OUTMUX2_00_a	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
164	sva/register_check/my_OUTMUX2_01_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
165	sva/register_check/my_OUTMUX2_other_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
166	sva/register_check/my_PULSEREG_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
167	sva/register_check/my_SETPORTINDEXREG_a	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
168	sva/register_check/my_SETPORT_a	assert	FORMAL_PROOF	infinite	COVER_PASS	1	yes
169	sva/register_check/my_SETREG_a	assert	FORMAL_PROOF	infinite	COVER_PASS	2	yes
170	sva/register_check/my_STROBE_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
171	sva/register_check/unknown_raddr_error_a	assert	FORMAL_PROOF	infinite	COVER_PASS	4	yes
172	sva/register_check/unknown_waddr_error_a	assert	FORMAL_PROOF	infinite	COVER_PASS	3	yes
173	sva/axi4lite_checker/axi_checker/cover_statements/AR_cover_back2back	cover	N/A	0	COVER_PASS	1	witness
174	sva/axi4lite_checker/axi_checker/cover_statements/AR_cover_no_waitstate	cover	N/A	0	COVER_PASS	1	witness
175	sva/axi4lite_checker/axi_checker/cover_statements/AR_cover_waitstate	cover	N/A	0	COVER_PASS	5	witness
176	sva/axi4lite_checker/axi_checker/cover_statements/AW_cover_back2back	cover	N/A	0	COVER_PASS	1	witness
177	sva/axi4lite_checker/axi_checker/cover_statements/AW_cover_no_waitstate	cover	N/A	0	COVER_PASS	1	witness
178	sva/axi4lite_checker/axi_checker/cover_statements/AW_cover_waitstate	cover	N/A	0	COVER_PASS	4	witness
179	sva/axi4lite_checker/axi_checker/cover_statements/B_cover_back2back	cover	N/A	0	COVER_VACUOUS	infinite	no
180	sva/axi4lite_checker/axi_checker/cover_statements/B_cover_no_waitstate	cover	N/A	0	COVER_PASS	3	witness
181	sva/axi4lite_checker/axi_checker/cover_statements/B_cover_non_okay_response	cover	N/A	0	COVER_PASS	3	witness
182	sva/axi4lite_checker/axi_checker/cover_statements/B_cover_waitstate	cover	N/A	0	COVER_PASS	3	witness

183	sva/axi4lite_checker/axi_checker/cover_statements/R_cover_back2back	cover	N/A	0	COVER_VACUOUS	infinite	no
184	sva/axi4lite_checker/axi_checker/cover_statements/R_cover_no_waitstate	cover	N/A	0	COVER_PASS	4	witness
185	sva/axi4lite_checker/axi_checker/cover_statements/R_cover_non_okay_response	cover	N/A	0	COVER_PASS	4	witness
186	sva/axi4lite_checker/axi_checker/cover_statements/R_cover_waitstate	cover	N/A	0	COVER_PASS	4	witness
187	sva/axi4lite_checker/axi_checker/cover_statements/W_cover_back2back	cover	N/A	0	COVER_PASS	2	witness
188	sva/axi4lite_checker/axi_checker/cover_statements/W_cover_no_waitstate	cover	N/A	0	COVER_VACUOUS	infinite	no
189	sva/axi4lite_checker/axi_checker/cover_statements/W_cover_waitstate	cover	N/A	0	COVER_PASS	2	witness
190	sva/axi4lite_checker/axi_checker/cover_statements/cover_interleaved_read_data	cover	N/A	0	COVER_VACUOUS	infinite	no
191	sva/axi4lite_checker/axi_checker/cover_statements/cover_read_next_data_in_order	cover	N/A	0	COVER_VACUOUS	infinite	no
192	sva/axi4lite_checker/axi_checker/cover_statements/cover_read_next_data_out_of_order	cover	N/A	0	COVER_VACUOUS	infinite	no
193	sva/axi4lite_checker/axi_checker/cover_statements/cover_write_control_after_data	cover	N/A	0	COVER_VACUOUS	infinite	no
194	sva/axi4lite_checker/axi_checker/cover_statements/cover_write_control_after_last_data	cover	N/A	0	COVER_VACUOUS	infinite	no
195	sva/axi4lite_checker/axi_checker/cover_statements/cover_write_control_before_data	cover	N/A	0	COVER_PASS	2	witness
196	sva/axi4lite_checker/axi_checker/buffer_restrictions/restrict_pending_requests	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
197	sva/axi4lite_checker/axi_checker/slave_assume/ARVALID_not_x_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
198	sva/axi4lite_checker/axi_checker/slave_assume/AR_after_lock_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
199	sva/axi4lite_checker/axi_checker/slave_assume/AR_legal_exclusive_burst_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
200	sva/axi4lite_checker/axi_checker/slave_assume/AR_legal_wrap_burst_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
201	sva/axi4lite_checker/axi_checker/slave_assume/AR_lock_continue_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
202	sva/axi4lite_checker/axi_checker/slave_assume/AR_lock_start_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
203	sva/axi4lite_checker/axi_checker/slave_assume/AR_long_burst_is_incr_and_not_exclusive_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
204	sva/axi4lite_checker/axi_checker/slave_assume/AR_no_4k_violation_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
205	sva/axi4lite_checker/axi_checker/slave_assume/AR_no_access_after_reset_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
206	sva/axi4lite_checker/axi_checker/slave_assume/AR_no_reserved_access_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
207	sva/axi4lite_checker/axi_checker/slave_assume/AR_request_signals_not_x_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
208	sva/axi4lite_checker/axi_checker/slave_assume/AR_stable_signals_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
209	sva/axi4lite_checker/axi_checker/slave_assume/AR_transfer_size_fits_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
210	sva/axi4lite_checker/axi_checker/slave_assume/AWVALID_not_x_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
211	sva/axi4lite_checker/axi_checker/slave_assume/AW_after_lock_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
212	sva/axi4lite_checker/axi_checker/slave_assume/AW_exclusive_burst_after_response_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
213	sva/axi4lite_checker/axi_checker/slave_assume/AW_first_wdata_in_order_of_control_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
214	sva/axi4lite_checker/axi_checker/slave_assume/AW_legal_exclusive_burst_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
215	sva/axi4lite_checker/axi_checker/slave_assume/AW_legal_wrap_burst_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
216	sva/axi4lite_checker/axi_checker/slave_assume/AW_lock_continue_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
217	sva/axi4lite_checker/axi_checker/slave_assume/AW_lock_start_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
218	sva/axi4lite_checker/axi_checker/slave_assume/AW_long_burst_is_incr_and_not_exclusive_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
219	sva/axi4lite_checker/axi_checker/slave_assume/AW_no_4k_violation_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
220	sva/axi4lite_checker/axi_checker/slave_assume/AW_no_access_after_reset_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
221	sva/axi4lite_checker/axi_checker/slave_assume/AW_no_reserved_access_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
222	sva/axi4lite_checker/axi_checker/slave_assume/AW_request_signals_not_x_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
223	sva/axi4lite_checker/axi_checker/slave_assume/AW_stable_signals_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A

224	sva/axi4lite_checker/axi_checker/slave_assume/AW_transfer_size_fits_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
225	sva/axi4lite_checker/axi_checker/slave_assume/BREADY_not_x_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
226	sva/axi4lite_checker/axi_checker/slave_assume/RREADY_not_x_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
227	sva/axi4lite_checker/axi_checker/slave_assume/WLAST_correct_both_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
228	sva/axi4lite_checker/axi_checker/slave_assume/WLAST_correct_ctrl_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
229	sva/axi4lite_checker/axi_checker/slave_assume/WLAST_correct_data_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
230	sva/axi4lite_checker/axi_checker/slave_assume/WSTRB_legal_after_ctrl_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
231	sva/axi4lite_checker/axi_checker/slave_assume/WSTRB_legal_before_ctrl_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
232	sva/axi4lite_checker/axi_checker/slave_assume/WSTRB_legal_with_ctrl_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
233	sva/axi4lite_checker/axi_checker/slave_assume/WVALID_not_x_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
234	sva/axi4lite_checker/axi_checker/slave_assume/W_data_beats_in_burst_limited_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
235	sva/axi4lite_checker/axi_checker/slave_assume/W_first_wdata_in_order_of_control_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
236	sva/axi4lite_checker/axi_checker/slave_assume/W_first_wdata_with_control_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
237	sva/axi4lite_checker/axi_checker/slave_assume/W_no_access_after_reset_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
238	sva/axi4lite_checker/axi_checker/slave_assume/W_request_signals_not_x_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
239	sva/axi4lite_checker/axi_checker/slave_assume/W_stable_signals_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
240	sva/axi4lite_checker/axi_checker/slave_assume/Write_interleaving_depth_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
241	sva/register_check/ARADDR_lowest_bits_0_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A
242	sva/register_check/AWADDR_lowest_bits_0_c	assume	FORMAL_ASSUMPTION	infinite	N/A	0	N/A

Figure 20: Quantify assertion coverage report.